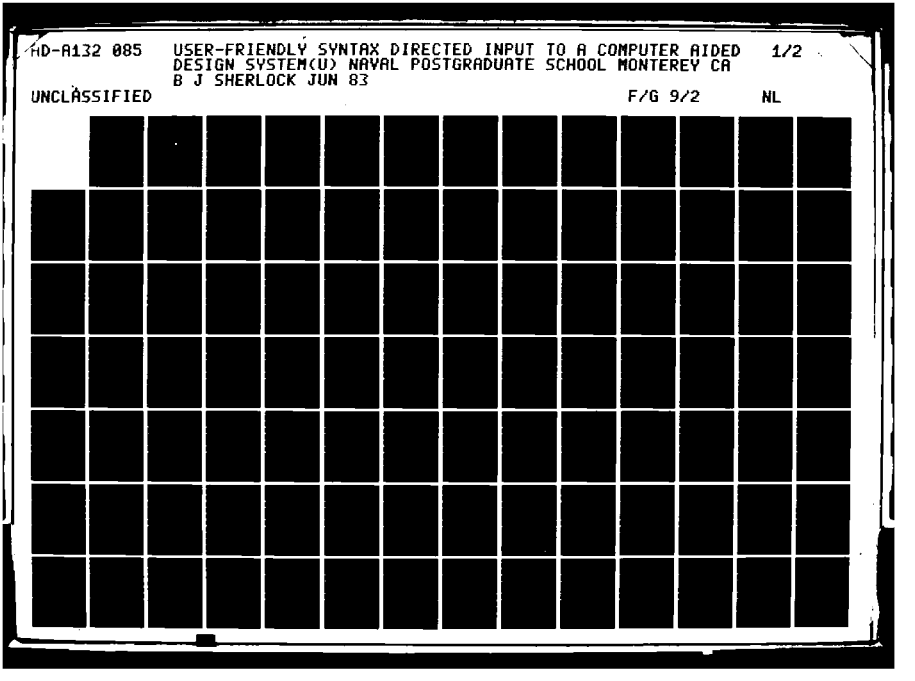
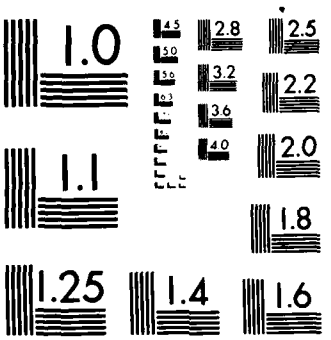
AD-A132 085 USER-FRIENDLY SYNTAX DIRECTED INPUT TO A COMPUTER AIDED DESIGN SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA B J SHERLOCK JUN 83 1/2

UNCLASSIFIED F/G 9/2 NL





NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

USER-FRIENDLY, SYNTAX DIRECTED INPUT TO A COMPUTER AIDED DESIGN SYSTEM

by

Barbara J. Sherlock

June, 1983

Thesis Advisor: Alan Ross

Approved for public release; distribution unlimited

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) User-Friendly, Syntax Directed Input to a Computer Aided Design System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1983 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Barbara J. Sherlock		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, Ca. 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, Ca. 93940		12. REPORT DATE June 1983 13. NUMBER OF PAGES 177
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, Ca. 93940		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

User-friendly, syntax directed editor, computer aided design, microprocessor control systems, man-machine interface, control system design automation, design environment, workstation, Computer hardware design languages.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes the development of a user-friendly, syntax directed input module of a computer aided design system. Color and dialogue are used to maximize user understanding and ease of interface, and minimize the opportunity for error. As a result, it is possible for the user to concentrate on the higher level aspects of the design, and allow the system to handle the routine details.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Approved for public release; distribution unlimited.

User-Friendly, Syntax Directed Input to a Computer Aided Design System

by

Barbara J. Sherlock
Lieutenant Commander, United States Navy
B.A., Wellesley College, 1972
M.B.A., Pepperdine University, 1975

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1983

Author: Barbara J. Sherlock

Approved by: Alan A. Ross
Thesis Advisor

Second Reader

Daniel K. Hsiao
Chairman, Department of Computer Science

Kneale T. Marshall
Dean of Information and Policy Sciences

ABSTRACT

This paper describes the development of a user-friendly, syntax directed input module of a computer aided design system. Color and dialogue are used to maximize user understanding and ease of interface, and minimize the opportunity for error. As a result, it is possible for the user to concentrate on the higher level aspects of the design, and allow the system to handle the routine details.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. THE PROBLEM

Computers are tools to be used by people to make completion of tasks easier and more efficient. Yet work is just now being done to make it easier for people to use the computers which are assisting them. With the variety of both computer systems and languages available it appears that it will be a long time before sufficient training is available to provide computer literacy for all who want or need it. The result of this is that two distinct groups of computer users exist - those who write programs for specific needs, and those who use the software packages produced by the first group.

But do users and user needs always fall into these two clear cut groups, or is there a need for a means for users to tailor software to their individual needs, or write their own without becoming programming experts? One no longer has to be an expert mechanic to know more about a car than how to drive it. Does one have to be a computer programmer in order to make full use of the resources and services of a computer? In the last decade, as computer cost has dropped and use of computers has become more common, various attempts have been made, and continue to be made, to make access to computers easier.

Two areas exist, in both technical and non-technical fields, where non-programmers are using computers extensively as means to deal with a large volume of available information in an interactive manner. The first is in use of query languages to access a database. The second is in computer aided design.

B. PREVIOUS ATTEMPTS AT A SOLUTION

1. Query Languages

One proposed solution to the above problem came in the examination of ways to simplify access to an online bibliographic retrieval system [Ref. 1]. Users of the system had to learn a different interaction language for each different computer system involved in the network. The questions were raised of whether the needs of the computer or the user should be given highest priority, and whether it would be possible to standardize or at least reach some compromise on access languages the user would need to know. Five solutions were proposed, covering a range of alternatives. In a totally non-user oriented approach, the user could be required to learn numerous different languages, one for each system. A second alternative was use of a standard language; however, no language standards yet exist. A third suggestion was the development of a system which could understand all languages. Due to the high cost and overhead involved, this was also considered infeasible. The two remaining viable alternatives, creating a system which appeared to use a standard language or a system which appeared to be able to understand all languages, required use of an intermediary to convert user languages to system languages. This 'uniformizer', transparent to the user, would receive a command from the user in the user's language, check it for syntax and semantics, and then convert it to the appropriate language for the object system. The reverse process would be followed for communications from the object system to the user. Thus the user would be able to deal with various systems using only one interactive language if so desired.

2. Computer Aided Design Systems

More technical examples of attempts to make use of computer resources easier have occurred in the field of computer aided design (CAD) This is a growing field, with the number of CAD systems both in use and proposed growing rapidly. Increasing use of CAD systems to design computer systems in particular is driven by the increasing complexity of the systems and the components of the systems being designed. Several examples of the use of computer system CAD systems, with various tools to make design easier, are the Programmer's Workbench, the S-1 Project, the Carnegie Mellon University Register-Transfer CAD System, and the automated design of control systems.

a. The Programmer's Workbench

The Programmer's Workbench (PWB), described in References 2 and 3, was developed in conjunction with computer aided design systems to make possible a software development system which would be general enough to be run on different host machines and to be used on different projects, thus being both machine and application independent. Problems arise in both computer aided design and in software development as a result of multiple types of computers, multiple languages, multiple databases, and poorly managed libraries. PWB, begun in 1973 at Bell Labs, was envisioned as a set of tools which could be useful throughout the design and maintenance of a computer system. PWB was designed to meet basic, continuing user needs for tools to: generate and modify manuals and other documents; create, edit, compile, execute, and debug programs; perform system generation, installation, and integration processes; test the subsystems and total systems and analyze test results; track system changes and test reports; monitor

system performance and be able to simulate and model system operations; convert data files and load databases when needed; and produce management reports and statistics throughout the development and maintenance of a system. The first implementation of PWB focused on the documentation, programming, and testing tools, assessing them as the most immediately useful aspects and the easiest for designers to learn.

The initial PWB thus consisted of five modules. The job-submission module handled job preparation, transmission from workstation to host, reports on status, and return of output. The module-control module kept track of revisions to programs, assigning identification codes and propagating changes to other related modules. A separate change module handled project wide changes and trouble and status reports. The documentation-production module provided macros for formatting documents such as the design specifications, test plans, and user manuals. Finally, the test-driver module provided means to simulate implementations on various controller/terminal combinations.

Use of PWB was found to have several benefits. Most obvious was reduced cost. It was cheaper to develop PWB with one set of tools than to work directly on each different host computer in the design process. The uniform programming environment produced by the PWB tools made training, documentation, programming standardization, and programmer retraining (for a new project or new machine) easier. Additionally, it reduced the conflicts arising during acquisition of new machines as a result of the differences between ideal online machines and ideal software development machines by providing an environment independent of the machine involved. Future work on PWB was expected in areas which would even further increase user convenience, such as using a standard programming language which PWB would then convert to code appropriate to the host machine.

b. The S-1 Project

The S-1 Project, using the Structured Computer-Aided Logic Design System (SCALD), is described in References 4 and 5. S-1 was an attempt to use graphics to make the designer's job in the computer aided design of large digital systems easier. SCALD was developed to reduce required design time "by allowing the designer to expres his design on the same level that he thinks about it" [Ref. 4: p. 271]. SCALD, written in Pascal, uses the Stanford University Drawing System (SUDS) for designer input to the system. Designers input a high level logical design drawing through a graphics terminal. A 'macro expander', working with a pictorial library of hardware components, repeatedly reduces the logical design to various stages of physical design, until data for the actual implementation and manufacture is produced. SCALD was used at Lawrence Livermore Laboratory to design the S-1, a 550-chip ECL-10K processor; however it has not been applied to general computer system design.

c. The Carnegie-Mellon University Project

The Carnegie-Mellon University Register-Transfer Computer Aided Design (CMU RT-CAD) System, described in References 6, 7, 8, 9 and 10, was developed as a system which could design computer systems from inputs containing functional descriptions rather than structural aspects. Designer input to the system includes a behavioral description of the system and the user's optimization criteria (for example, time or cost). Also included is a database of available hardware components which can be permanently stored and updated, rather than reentered each time. The system then transforms this input into a form readable by the 'allocator', and attempts to construct the desired

system from components available in the hardware library, within the constraints given by the designer. The CMU RT-CAD system has been used in system simulation, design logic verification, and actual hardware design. It can be adapted to different design styles, such as centralized, distributed, or pipeline systems.

d. Automated Design of Control Systems

Attempts have also been made to use computer aided design in the design of control systems, again beginning with fuctional descriptions. One model for the automation of real-time controller design, described in References 11 and 12, uses functional level input in terms of pairs of contingencies and tasks, a function to determine when a contingency exists, and a procedure to carry out the resulting required task. Also included are environmental data (controller input and output), design criteria, and designer and project information. Data entered by the designer is converted into a 'primitive list' description, and then compared with possible realizations in a hardware description library until a match is found. Implementation of the model includes the action linking the primitive list to the hardware realizations and the development of a monitor to sequence the contingency test and task execution pairs.

Another model, LOGE-MIR [Ref. 13], uses pictorial flow diagrams as the means to input basic design information, since flow diagrams are a common design format. Because the design system is limited to control applications, it is possible to establish a strict problem specification format using flow diagram symbols. The flow diagram is then converted into an intermediate language and general optimizations are done. In the final step, the intermediate language is combined with an interpreter for a specific

microcomputer to produce the design output. This two step transformation makes it easier to adapt the design system to new microcomputers, since the design input can remain the same and only the interpreter must be rewritten. Another advantage of this system is in having the system, rather than the designer, write the design software, thus reducing the opportunity for errors.

C. THE CURRENT SITUATION IN COMPUTER AIDED DESIGN

In discussing the CMU RT-CAD system, Barbacci [Ref. 9] pointed out that designers should not have to do "repetitive, time consuming tasks" such as generating detailed design information, monitoring changes in design documents, checking systems for electrical, logical, or physical compatibility, or developing manufacturing information. Rather, as technology evolves, with primitive components becoming more complex and the rate at which new components are introduced increasing, designers must be able to design at a higher level, and must be able to design faster. Ross, in commenting on computer aided design systems in general, and means for design of control systems specifically, pointed out that systems had to be attractive to be used, and should "perform a helpful part of the design task, must present the results in a useful format, and must be easy to use" [Ref. 11: p.87].

However, despite the good intentions, a gap exists between the designer of a CAD system, with his or her assumptions and expectations regarding the user and the user's needs, and the actual user of the system. As discussed in References 14, 15, 16 and 17, CAD systems exist to aid the designer in the design process, not to automate it completely. CAD system users are not computer programmers, hence their input should be in something close to

their design language, not in a programming language. In addition, it appears that each individual system is designed for a unique need, and with a unique language, thus limiting the user's flexibility. The tools which thus result from these false assumptions require users to think in a new way, or learn a new language, and are more of a hindrance than a help. Users are expected to be not just engineers or designers in a particular field, but also computer programmers with time to learn a special purpose language. As with bibliographic query languages and procedures, a need exists for some sort of adaptability in computer aided design systems, and the languages which they use.

D. THIS PROJECT

In an effective design environment, the designer can focus on design rather than spending time on implementation details and repetition of tedious tasks. The purpose of this project was to create a design environment in which, with a minimum of training in areas outside those actually required for design, a user would be able to use the available tools to enter required data. Emphasis was placed on designing a workstation which operates in such a manner that the user is not required to learn an additional programming language.

The model developed for automated control system design served as a vehicle for this project. Tools were still needed in this model to create a workstation for design data input and convert that input to the primitive list format. This project focused on the man-machine interface of the model, the means to input the design data.

The goal in designing and implementing this design environment was to build a workstation which removes the need for the designer to learn a specific language with which to

enter the data. Additionally, the system should present a user-friendly approach to the interface. Finally, the system should produce an output of the data entered in a user readable format which is ready to be converted to the primitive list format.

Chapter Two will summarize the development of hardware description languages and the human factors which must be considered in developing a computer aided design system. Chapter Three will examine the implementation decisions made regarding design of the system. Chapter Four discusses the implementation of the system, and Chapter Five considers the conclusions which can be drawn and recommendations made for future work.

II. BACKGROUND

A. HARDWARE DESIGN LANGUAGES

A critical requirement for a computer design system is a hardware design/description language. Over the past 15-20 years, computer hardware description languages (CHDLs) have gone from being useful to being a necessity as technology and design components have become more complex. Both Chu [Ref. 18: p. 19] and Van Cleemput [Ref. 19: pp. 554, 559] see computer hardware description languages as being able to provide:

accurate communication among designers and engineers;
precise, concise, and convenient documentation;
a means for system simulation and verification; and
a means to automate system design and realization.

Hardware design progresses through a hierarchy of stages, from system level through register-transfer and gate levels to circuit level and logic detail. Design can be approached from either a top-down or bottom-up perspective [Ref. 20: p. 377]. As a result of this, some CHDLs deal with only one or two levels in the hierarchy, while others attempt to deal with all levels in the process. Additionally, the earliest CHDLs described systems on only the most primitive level, and dealt directly with implementation. As systems became more complex, higher level description languages developed, much as higher level programming languages have been formulated in the past decade.

1. CDL

CDL, described in References 21 and 22, was introduced in 1965 as "an Algol-like computer design language" to provide a language which could be used as a standard design language, making possible both communication among designers and well-documented designs. Additionally, it was expected that CDL could be used to test and debug designs, simulate and evaluate performance, and make possible automated design and design of more complex machines. CDL met the basic requirements set forth by Chu that it be like natural language, concise and precise, translatable into boolean equations, able to be modified as the technology it was describing changed, and be able to represent and manipulate data, control, and timing signals in binary format.

CDL descriptions, 'sequences' (which are actually microprograms), are used to define the implementation of an algorithm, and follow a standard format. First is a name for the sequence, followed by declarations of registers, subregisters, memory, terminals, and operations. Statements are the final component of the description. Each statement is labeled, either with a boolean label or a name if it is the first line in one of the declared operations. Compound statements share a label and are expected to be executed within one clock period. Statements are not executed unless their boolean labels are true; this provides the control structure. Thus, as shown in Reference 23, a typical segment of a sequence in CDL would be written:

```
/fetch*p3/        IR <- MDR,
boolean label     statement
```

While CDL makes it possible to name basic circuits, registers, inputs, and outputs, and write conventional arithmetic and logical operations, it still remains a fairly low level description language. It retains the ability to

describe bit level actions such as a shift, but would be difficult to use at higher than a register-transfer level. However, an attempt has been made to extend CDL into MDL, Microcomputer Design Language [Ref. 24], which will make possible the hierarchical description of a system through use of level numbers. MDL can be used with SDL-1, Software Design Language 1, to show hardware and software interaction at a given level of a system.

2. AHPL

A Hardware Programming Language (AHPL), discussed in References 25, 26, 27, 28 and 29, was developed as an attempt to design a functional level, rather than a register-transfer level language. AHPL was based on the APL programming language, with APL's vector notation making it possible to more easily deal with entire registers rather than individual flip-flops, while at the same time being able to identify individual bits or segments of registers. AHPL attempted to partition a system into control sequences and data registers, and provide an alternative to circuit diagrams and state tables which became more involved as systems became more complex.

The standard format of AHPL consists of a declaration of inputs, outputs, and registers, followed by statements. Rather than CDL's boolean labels, statement lines are labeled with sequential numbers to correspond to the rows in an equivalent state table. All data transfers on one line are expected to occur simultaneously. Statement lines are executed sequentially, unless followed by a statement to branch to a different line. This produces a 'control sequence' of transfers and branches. As a final step, names can be assigned to transfers to produce a "combinational logic subroutine" [Ref. 25], with the name used instead of the detailed notation. However, this is the

closest AHPL comes to being a functional rather than a register-transfer language.

3. DDL

Digital System Design Language (DDL), described in References 30, 31, 32 and 33, was designed to fill some of the design language needs not met by languages such as CDL and AHPL. It was developed to meet several goals, among them that it be useful at all levels of the design process, from block structured architecture to gate level activity, that it not be limited in application to a specific hardware technology, that it be able to be used as a source language in automated design in the future, and that it provide documentation which matched the actual system organization.

The DDL system model consists of one or more automata (finite state machines) which control data facilities. Data facilities can be either private, if controlled by one automaton, or public, if controlled by more than one. Each automaton can be divided into segments. Repeated transformations of this initial system description, or a design initiated at any intermediate level, will ultimately result in a final system description consisting of logic equations. Dietmeyer describes DDL as "not well suited for describing extremely abstract models....[or] for presenting fabrication and technological details. It is intended to bridge the gap" [Ref. 32: p. 38]. Later revisions of DDL add a module construct to maintain modularity in each transformation output and thus make a top-down design methodology using DDL more feasible.

4. SDL

Also based on the concept that hardware design goes through stages from higher to lower level, SDL [Ref. 20] was developed with the objective of providing an "accurate

representation of structural information useful over all levels of the design process" [Ref. 20: p. 378], with a designer able to use SDL to map from higher level hardware primitives to lower level implementations.

A structural information description in SDL consists of a name, external connections (input/output), component types, and interconnections among the components. Additionally, each description must have a purpose or possible use statement, and an indication of which level of the design hierarchy (system, register-transfer) the description refers to. Level and purpose statements are related, and each purpose and level pair will use a library of resources which are available at the next lower level in the design hierarchy.

5. "S"

Another language which uses an approach similar to that of SDL is "S" [Ref. 34]. Design using "S" begins with a natural language description, since this is where a designer actually begins the design process. Also, this is easier to understand than some of the formal design and description languages. "S" makes it possible to process the description and add levels of increasing syntactic and semantic restrictions until all ambiguities have been removed. At this point the computer can examine implementation possibilities. Basic structure of a description in "S" includes a declaration section, with input/output variables and their types, conditions ('inhibitors') of the system, and a section for process description.

6. ISP

Instruction Set Processor (ISP) notation, described in References 9, 35, 36, 37, 38 and 39, and later modifications ISPI and ISPS, were developed as part of a set of

languages intended to be able to provide a description of a computer from the top of the hierarchy, the system level, down through the lower levels. A fifth level, the programming level, unique to computers, was added to the traditional design hierarchy. This programming level is located between the system level and the register-transfer level. ISP is used to describe this level, that of the machine's instruction interpretation cycle, where memory is accessed and operations are performed, in terms of the next lower level, the register-transfer level. ISP is thus the interface between the programming level and the register-transfer level. Symbolically a register-transfer language, but more general than most, it describes what occurs at the register-transfer level, but not how it occurs. ISP excludes timing data and other details of that sort.

In format, ISP resembles other register-transfer languages. Declarations include memory, processor and registers, any external connections, and data types available. The remainder of the description is devoted to the instruction interpreter, describing the steps in the fetch-execute cycle and each instruction in the instruction set.

ISP has been used extensively in the Carnegie-Mellon University RT-CAD system described in Chapter One as a tool to describe digital systems, simulate and emulate systems, synthesize hardware and software descriptions, and verify designs. This has revealed some limitations in ISP [Ref. 38]. One is a lack of a formal semantic definition of the language, which has led to variations in the format of descriptions in ISP, and the use of ISP to describe only part of a system. Another limitation is the lack of a means to handle concurrency, timing data, and interconnected processors in ISP.

7. CONLAN

As the number of computer hardware design languages multiplied, it was decided that the need existed for some common base from which to develop future languages. As a result of this decision, a working group of the designers of AHPL, DDI, AND ISP, among others, was formed in the middle 1970s to develop CONLAN (CONsensus LANguage). Discussed in References 40, 41, 42, 43 and 44, CONLAN was intended to provide for greater acceptance of hardware description languages through providing:

> a common formal syntactic and semantic base for all levels and aspects of hardware and firmware description, in particular for descriptions of system structure and behavior...
>
> a means for the derivation of user languages from this common base...
>
> [support to] CAD tools for documentation, certification, design space exploration, [and] synthesis..... [Ref. 40: p. 210].

It was expected that languages developed from Base CONLAN would be designed for a specific set of tasks and have a limited scope, be easy to learn and simple to use, and have a clear semantic relationship to other languages also developed from Base CONLAN.

Base CONLAN is defined by three items, a set of object types and operations, syntax, and a computation model. It is defined with Primitive Set CONLAN, a language used only for defining other languages derived in CONLAN. Once a language has been derived from Base CONLAN, its use as a computer hardward design language follows a standardized format. Each description serves as a module by itself. A description is begun with a name, followed by a listing of assertions of static conditions and dynamic constraints. Then the items from an external segment library, if any, are

listed, and internal and external variables are declared. Finally, the behavior of the model is described, using activities and functions, with 'end' indicating the end of the description.

8. CSDL

While there have been numerous register-transfer languages developed, and attempts to develop languages which can trace design through the entire design hierarchy, there have been few languages written which actually define a system in terms of its functions rather than its structure. Additionally, most computer hardware design languages describe operations which are carried out in a definite sequence, rather than concurrently (programming languages have also followed this approach, with Ada being the first major language in which concurrency was included in the original design). An attempt was made to meet both of these requirements in the development of CSDL, Control System Design Language [Ref. 45], utilized in the control system design automation project discussed in Chapter One. The goal of CSDL was to simplify problem specification and provide a means to automate hardware selection and software production in control system design. One way to accomplish this was to develop a language in which the designer had only to provide the behavior of the control system, what it did given certain inputs and outputs, rather than how it actually did it, or what hardware components were used.

CSDL descriptions have four major sections. First, as an aid to good documentation, is an identification section with the designer's name, the date, and the design's title. This is followed by the environment, the input and output variables to and from the controller. Third are the contingency lists, the key to what makes this a design for multiple concurrent activities. Contingency/task pairs

indicate the relationships between a contingency, a task which must be carried out when that contingency occurs, and ongoing physical time. The final section in CSDL is the procedural one, with functions which return boolean values defining contingencies and sampling the environment, and tasks performing actions on elements in the environment and making changes in it.

While BNF notation for CSDL has been written, and CSDL has been used as documentation for control system design automation research projects, a compiler has not yet been written for it and thus it has not yet been used as source code input for automated design.

B. MAN-MACHINE INTERFACE

1. Dialogue

A second critical need in a computer aided design system is for a design environment, similar to a programming environment. The important consideration in this is not the designer or the computer individually, but their interface, since

> human factors engineering...is concerned with ways of designing machines, operations, and work environments so that they match human capabilities and limitations [Ref. 46: p. 8].

In an interactive computer system, the need is for system components with which the user will be

> able to engage in a man-computer dialogue so designed that he is essentially unaware of the computer or the medium in which the dialogue is conducted [Ref. 47: p. 49].

Dialogue is defined as "nonprogramming communication with a terminal" [Ref. 48: p. 53], and is what makes a computer system interactive. Responses from the system in a dialogue can be placed in one of three categories, those which never change, those which change periodically, and those which change on a real-time basis as the dialogue is conducted. Dialogues in the first two categories are generally written by the programmer and stored in memory. While real-time dialogue can be close to natural language, it requires a great deal of software, including both a dialogue translator or processor and one or more dialogue files, as implemented in systems described in References 49 and 50.

Dialogue style can be one of four types, based on two independent characteristics, whether the user of the system guides the the interchange, and whether the user is limited to given choices in reply or can make a free response [Ref. 51]. At the ends of the continuum, dialogues which the system guides, with limited choice response, are faster, with fewer entries required for routine actions. User guided free response dialogues provide maximum flexibility (and maximum opportunity for error) for experienced users. Martin [Ref. 48], in what has become a classic in interactive system design, categorizes 23 techniques for dialogue design, with factors which can help determine the most effective type of dialogue for a specific task and environment.

Smith [Ref. 52] provides the two essential properties of the resulting system. It should be effective, able to accomplish something significant in a 'reasonable' amount of time, and it should be simple, although its simplicity will be inversely proportional to the complexity of the tasks it performs. In doing this, the system should be self-explanatory, self-helping, easy to use, able to anticipate the user's actions, and able to adjust to the skill level of the user.

Reference 53 defines three components of a user interface: the user's conceptual model of the actions going on, the command language used by the user to communicate with the system, and the information display used by the system to communicate with the user. Increasing the significance of the graphics factor, and with some reorganization, Newman and Sproul list four components of the user interface:

> User Model - the model which the user has of the information involved and how it is being manipulated and processed. This component underlies the other three.
>
> Command Language - how the user operates on the information and the model.
>
> Feedback - how the system provides the user with information and assistance in running the system and using the model.
>
> Information Display - the screen display of the state of the model and the information being manipulated. This can be used to confirm that the user's perception matches the state of the system [Ref. 54: pp. 445-448].

Of critical importance is the fact that the user's model is a mental image, not something in writing. It is something with which the user must be comfortable, and be able to become familiar. Thus, to make the system easier to learn and, at least intuitively, easier to understand, the system should use concepts and language familiar to the user.

2. The User and User Psychology

The user, and the user's psychology, are some of the first factors which must be considered in designing an interactive system from which the user will be able to develop a useful and appropriate model.

Eason and Damodaran [Ref. 55] consider as significant factors in analyzing computer interfaces: job related user attributes such as the training provided and the

relationship between the user's total job and the computer system; user requirements of the system, including psychological needs; variables related to the task such as structure and information handling; and individual user personality characteristics. Martin [Ref. 48] has determined six basic criteria with which to categorize terminal operators. First, they can be either dedicated or casual, working constantly at the terminal or only occasionally. The second consideration is their level of programming skill. Third is their intelligence level, specifically their short term memory span and ability to think logically. Fourth is their level of training and skill with regard to the terminal they are using. The fifth factor is whether they are active operators, taking the initiative in the task, or passive ones, following the direction of the computer. Finally, are there intermediaries between the user and the system, or is the user able to deal directly with it? Martin gives additional consideration to the case of the totally untrained operator.

Five potential blocks to maximizing user involvement with an interactive system exist [Ref. 56]. Users can become bored with a system if pacing of screens is too slow or response time allowed is too great. Unexpectedly long delays without system response can lead to user panic due to fear that there is something wrong in either the system or the program being executed. Frustration can result from being unable to easily, effectively, and efficiently communicate with the system and safely recover from unwanted or unexpected actions. Inability by the user to formulate an appropriate user's model, caused by either excessive detail or lack of structure in the system, can lead to confusion on the part of the user. Finally, discomfort can result from shortcomings in the physical environment, such as poor lighting or lack of sufficient work space.

Related to some of these potential problem areas are some basic human psychological characteristics. First, as noted by G. A. Miller [Ref. 57], the factor seven, plus or minus two, is a recurring item in human short term memory research. Whether data is considered as individual items, or recoded into 'chunks' as, for example, letters into words, or individual numbers into a telephone number, humans are generally able to recall only seven chunks or make accurate decisions among no more than seven choices. Exceeding seven (plus or minus two) units of data in providing information can lead to information overload on the part of the user.

While Miller has shown that people can handle more data if they can group it into chunks, human ability to handle pictorial information can also be a factor in increasing the amount of information a user can handle when working with an interactive system. Haber and Wilkinson [Ref. 58] point out two principles in human visual perception: the human vision system is designed to capture the total situation perceived, and the system will try to interpret all components as part of the scene. Thus, visual displays can be used to convey not just bits of information, but also the structure of the information, to the user. With this approach, there will be less need for the observer to consciously interpret the organization of the information presented, or try to group its individual components into chunks.

This human need to organize information also has an effect on how people keep track of their activities [Ref. 59]. Just as people organize bits of information into chunks, they organize their activities into 'clumps', sequences of steps which lead them toward a goal or accomplishment of a task. Completion of one of these clumps of actions leads to 'closure', indication that a job, or

subject, has been finished and can be crossed off one's mental 'to do' list. When working in an interactive system, whether with other humans or machines, one expects not only an internal feeling of closure upon completing a clump of actions, but also a feedback response from the other part of the system, either during or at the end of the activity. Depending on the specific type of action, this response is expected within a certain, limited, amount of time. Delay can lead to frustration and lack of continuity of thought, while closure and response brings the opportunity to clear short term memory and move on to a new thought. Response time delays are more acceptable once closure has been reached than they are before, since short term memory can store data, such as a phone number, for only a limited amount of time. As a general rule, delays over two seconds are acceptable once closure has been reached (waiting for the ringing after dialing a telephone number), but not before (waiting for a dialtone so a telephone number just looked up can be dialed). The decrease in efficiency resulting from response time delays is not a linear relationship. Rather, a delay produces a sudden drop in user efficiency, followed by a leveling off for a period, what R. B. Miller refers to as "psychological step-down discontinuities."

As a guideline to dealing with the characteristics of designers using CAD systems, Spence and Apperley [Ref. 60] provide a checklist of items to be considered, based in part on the preceeding psychological factors:

Short term memory is limited; an interruption will lead to forgetting.

Psychological closure is needed; provide an indication to the user that a task is complete.

Consider computer response time; allow time for the user to see the result of an action.

Respond to control actions; predictability is needed.

Consider human pattern recognition skills; make use of them in data interpretation.

Use words appropriate to the task; users need familiar terminology.

Martin [Ref. 48] proposes that, when designing a system, it be considered on three levels: functional (how to use both human and machine capabilities most effectively), procedural (how to organize the system), and syntactical (how to handle communcations between the user and the system). Ginsburg, quoted in Reference 61, points out the critical importance of the user, and the user's perspective, in the effectiveness of an interactive system:

> Nothing can contribute more to satisfactory system performance than the conviction on the part of the terminal operators that they are in control of the system and not the system in control of them. Equally, nothing can be more damaging to satisfactory system operation, regardless of how well other aspects of the implementation have been handled, than the operator's conviction that the terminal and thus the system are in control, have 'a mind of their own,' or are tugging against rather than observing the operator's wishes. [p.12]

This then leads to consideration of the remaining three components of the interface: the command language, feedback, and the information display, the more tangible components which make up the system hardware and software.

3. Command Languages

Four design issues exist with regard to the command language in an interactive system [Ref. 54: pp. 451-458]. First, how many modes should there be? More modes lead to greater complexity and thus more errors. Second, a selection sequence for the use of commands and command parameters needs to be determined. In this, consistency is important. Third, an abort mechanism, through which a user can terminate an action begun, must be provided. Finally, an error

handling mechanism must exist. A wide range of command language styles are available, including keyboard dialogue with screen prompts (flexible but inefficient), a keyboard command language with standard command words and a simple syntax (requires more user skill), use of function keys (faster, simpler, but less flexible), and menu driven (more flexible, fewer errors, uses more screen space).

Some guidelines have been proposed for designing a command language [Ref. 53: pp. 6-7]. The language should be consistent: each key should always have the same, or at least an analogous, meaning in the language. Second, a minimum effort should be all that is required from the user to enter the commands, with the commands most frequently used being the easiest to enter. Finally, use of only one mode is encouraged. However, if more than one mode is necessary, any one command should have the same meaning in each mode in which it can be used.

There is not yet agreement on whether command languages should vary depending on the skill level of the user. Mozelco [Ref. 62] proposes five levels of user motivation and expertise, and corresponding types of command languages. The five user levels are: the user who is learning the basics; the user who wants only to use the system to get acceptable results with a minimum of learning; the user who wants to be able to use the system more independently; the user who wants to learn the more subtle features of the system; and finally, the user who wants to get the highest possible quality results. The five corresponding levels of command language range from simple languages using multiple choice questions and tutorials to more complex languages which use function keys and an editor to create command sequences for later use. While creating several languages for one system is costly in terms of both time and money (and memory), Mozelco thought the approach worth considering.

Walther and O'Neill [Ref. 63] came to related conclusions. Subjects with different skill levels, working on either teletype (TTY) or cathode ray tube (CRT) terminals were able to work with either a flexible or inflexible text editor. Use of the flexible language, in which users were able to abbreviate, omit, or change command words, among other options, did not consistently improve performance. User attitudes, which were effected by all three variables (terminal type, language type, and skill level), had an effect on performance, as did each of the variables individually. With the inflexible language, even users with little experience made few errors. With the flexible language, for all but those with the most experience, a less positive attitude led to more errors. If job completion time were the only consideration, it appeared from the data that it would be appropriate to provide more experienced users with a flexible command language to speed their work. However, if syntax errors were also a consideration, experience alone was not enough on which to base the language decision. Attitudes, something more difficult to measure, and more likely to change over time, also had to be considered. This is consistent with Mozelco's approach of providing several levels of command language and leaving the decision up to the user. Unfortunately, the question remains of whether this approach is worth the cost.

One of the most important components of a command language is a 'help' facility. Shneiderman points out that, as with levels of language, user experience is not the best way to determine how much help to provide, "since even experts may forget or be novices with respect to some portions of a system" [Ref. 61: p. 17]. He recommends that the user control the level of help provided through the help request made to the system. Relles and Price [Ref. 64] expand on this, focusing on the needs of both the user and

the programmer of the help facility. Help requests should be simple, concise, and consistent, and the user should be able to maintain the current task during the help session. The help provided should be specific to the current context, and precise, providing only the information needed. Additionally, help messages should be polite, in the user's language, and should include information on what the user should do next, or how more detailed information can be obtained. From the programmer's point of view, it should be possible to specify the help routines at a high level of abstraction, separating writing the text of the message from writing the rest of the code. Finally, messages should be easy to write and easy to modify.

4. Feedback and Error Handling

Feedback to the user is provided with regard to the command language system (a prompt, response to a command, or error message), the application database (the actual task being performed, such as text editing or CAD), and the display terminal (cursor movement or character echo). As pointed out earlier, timely feedback is critically important.

Perhaps the most important feedback is that regarding errors. User errors can be caused by user overload (sending too much information to the user at once), user boredom or lack of motivation (lack of a timely response, or performance of only passive operations), or inadequate instruction and guidance to the user regarding the system. Whatever the cause, it is important that an error message be given as soon as possible after the error has occurred. Error messages should be short and to the point, but should not be negative or embarrassing to the user. In addition to stating that there has been an error, messages should contain useful information, such as where

the error occurred and how to fix it. Finally, it should be possible to fix the error with a minimum amount of work.

5. Display

Two issues must be dealt with in the area of information display: what will the overall layout be and how will objects and information be represented. While specific layout can be determined by deciding exactly when the user requires what information, five general areas are needed: a main work area, an area for local editing and input, an area for system status indicators, a diagnostic area for error messages, and a menu/information area [Ref. 51]. Blinking, highlighting, or reverse video can be used to draw attention to certain areas. Color can be used to lessen apparent clutter on the screen, emphasize certain areas or items, confirm entry of data into the system, or make identification of some items easier for the user. However, if color is to be used, consideration must be given to the likelihood of colorblind users [Ref. 48].

Menus can be used as part of the display as a continuous source of information to the user, with color and position used to group related items. To decrease the amount of screen space required, a hierarchy of smaller menus can be used. Additionally, consideration has been given to the use of a hierarchy of dynamic menus which change as the dialogue progresses [Ref. 65]. With this approach, three menus are available on the screen at one time: a constant menu of most frequently used commands, the dynamic menu, and a menu of menus. While dynamic menus are possible, it is not yet known how feasible a system of this sort would be.

Based on the above information on computer description languages and human factors, the design procedures followed and decisions made will be examined in Chapter

Three. Implementation details will be discussed in Chapter Four.

III. DESIGN

A. PRELIMINARY ASSUMPTIONS

The underlying concepts guiding the design of the system follow those stated by Shneiderman:

> build systems that behave like tools; and
>
> recognize the distinction between human reasoning and computer power [Ref. 67].

Martin echoes these concepts: "*Do not try to make the computer compete with man* in areas in which man is superior" [Ref. 48: p. 7]. The intent of the system is, as noted by Thomas [Ref. 68], to make possible the synthesis of designer and computer, to provide (a component of) a design system which will aid the designer in top-down design of a control system, with behavioral descriptions as input.

Several preliminary assumptions were made regarding the designer and the system. First, the designer has some general programming knowledge (i.e. is familiar with the concepts of block structured languages and top-down design) but does not necessarily know any specific language well. Second, he or she will also be familiar with the basic components of a control system - the contingency/task pairs, the environmental variables, the design criteria, and the task and contingency subroutines - and the information included in each of these components. Third, the designer will use the system occasionally rather than continuously. Finally, for its initial implementation, the entire design will be entered and a realization produced at one time.

B. DESIGN CRITERIA

Design criteria were considered in two categories: what the system should be able to do, and how it is organized to do it. The system should (in order of decreasing priority):

not require the designer to learn a specific design/programming language;

minimize the amount of repetitious, tedious work required from the designer;

minimize actions taken by the system implicitly or by default which could lead to hidden errors; require explicit confirmation from the designer whenever possible;

allow the designer to make changes to already entered data without having to reenter all of the data;

be able to respond to different levels of designer expertise.

Regarding system organization, it should be:

fully modularized;

implemented as a minimal set, with the ability in the future to:

add new constructs and/or subroutines to the design language;

add new types and categories of design information;

modify the format in which the design data is stored, both during execution and when written to a file.

C. PRELIMINARY DECISIONS

Several basic decisions were necessary prior to actual project design.

1. Computer System Selection

The primary consideration for system selection was that the system be available for dedicated work. However, it was also important, since an interactive system was to be designed, that the system be able to meet most, if not all of the requirements generally provided for interactive systems. Martin [Ref. 48] proposes, as considerations when selecting components for interactive systems, that there be means for input by both the user and some document source; means for output; a display screen with appropriate capabilities, such as color; tools to provide the desired or needed level of security; and tools for error control. Irby [Ref. 66] expands on the required screen capabilities, giving five necessary items: the programmer must be able to write to arbitrary positions on the screen, be able to map conceptual display primitive operations to device primitive operations, be able to highlight text and remove highlighting, and have a coordinate input device which can be tracked on the screen. It was also considered desirable, though not necessary, that it be possible to scroll the screen, use different character fonts, and have a terminal intelligent enough to be able to reply to questions from the display terminal interface. The VAX/VMS system, with GIGI (Graphics Image Generator and Interpreter) terminal and monitor [Ref. 69] was available at NPS, and, since it met most of the above requirements, was selected as the system upon which the project would be implemented.

2. Programming Language

Of critical importance was the ability of the language selected to provide the necessary data structures, primarily records as elements of linked lists. Pascal, Ada, and 'C' were considered. Pascal was selected because, in

addition to providing the necessary data structures, it is relatively self-documenting, a trait of some importance since it is expected that future researchers will continue work on this project. Also, Pascal is a well tested and operational language.

3. Design Methodology

Large software projects require some organized approach to design and implementation. While 'step-wise refinement' is a frequently used term with regard to software design, there are numerous ideas and theories about exactly how to carry it out.

One of the more commonly used methodologies is that developed by Constantine. The 'structured design' methodology [Ref. 70] proposes that systems be considered as an input/process/output sequence during design. Attention is first focused on what functions the system will perform. Then these functions are examined in terms of their input and output components, and what transformation must occur between the two. This process of determining function, decomposing into input/process/output, and then analyzing each of these modules in terms of function, is repeated until the system has been completely decomposed. Only at that point does implementation begin.

Structured design may be more commonly used than other design methodologies because it is easier to conceptualize and use, focusing as it does on input and output. Additionally, its documentation appears similar to flowcharting, although there are significant differences. Finally, it can be used in conjunction with IBM's Hierarchical Input-Process-Output (HIPO) design documentation technique.

Another software design approach is suggested by Parnas in References 71 and 72, where he emphasizes concentrating on the factors most likely and least likely to change, and hiding arbitrary implementation decisions. With this approach, specific abstract data structures are hidden within one module, with module interfaces revealing a minimum amount of information about design decisions. As a result of this approach to decomposition, it is possible to break a system down into levels of subsets, with the lowest level a basic, minimum subset. Modules can be placed in a loop-free 'uses hierarchy' of modules which use lower level modules.

This information hiding and hierarchy of users criteria for system decomposition, placing emphasis on what in the system is likely to change in the future, facilitates system change at a later date. For this reason it was decided that an attempt would be made to use a combination of the two approaches above. There will be attempts in the future to modify or expand the implementation of this project, and consideration was given to making that task as simple as possible.

In line with the design guidelines given by Parnas, consideration was given to where change was most likely to occur in the design system being built. Three areas were considered: the format of the data entered, the content of the data entered, and the command language used to enter the data. It was decided that the content of the data which would be entered would be the least likely to change significantly, since control system components are fairly standard. However, the format of the data, in both the source language and primitive list realization, was likely to change. Additionally, it was likely that the command language would expand to allow for different levels of user expertise.

D. SYSTEM COMPONENTS

1. Design Language

Reference 38, in analyzing ISP, provides guidelines for the developemnt of the ideal behavioral hardware description language. It should:

provide abstraction facilities with which to add more primitives;

make it possible to specify behavior without structure;

support structured programming constructs;

make it possible to describe application specific information;

have the capacity to express concurrency;

have the capacity to describe multiple functions;

have the capacity to express synchronizing primitives explicitly; and,

have a formal semantic definition of language operations.

Reference 73 adds the further guidelines that a design language should be suitable for a variety of design applications, be easy to learn and document through uniformity and brevity, be suitable for use with interactive graphics, and be adaptable to more complex designs.

In the specific design language to be used, a means was needed to input designer information (name, date, design name, comments), environmental information on variables, contingency/task pair information, criteria information, and the actual functions and procedures for the contingencies and the tasks. Additionally, the language had to be able to store the data during run-time operations in a format through which the data could be easily transferred to the IADEFI (identification, application timing for C/T pairs, design criteria, and environment) file and the primitive list of subroutines used in realization of the design. A

final requirement was that the design also be able to be documented by an easily readable source file.

While Biehl and Dietzinger [Ref. 13] used flow diagrams to enter data, the development of higher level languages such as Pascal and Ada is leading to the use of languages rather than flowcharts to explain program design in the United States. Additionally, use of a high level language in design is closer to the algorithmic process, more like natural language, and makes the programming of software easier and more efficient [Ref. 74: p. 431]. For these reasons, the decision was made to use a language, rather than pictorial flowchart representations, to enter data.

A block structured programming language was necessary since without it the control system tasks and functions could not be properly entered. With regard to the remainder of the design data, the possibility of having no formal design language, but instead entering data directly into the primitive list, was considered. However, software documentation is becoming increasingly important, and it was determined that some input record in a more readable form than the primitive list was needed.

As the design of the project progressed, it was determined that two languages, rather than one, needed to be developed. The first language was needed to, as a minimum, enter the functions and procedures involved in the contingencies and tasks, and store data during execution. This can be considered a run-time language, and is written in terms of the data structures used during operation of the design system. The second language needed was a more formal written language, to be considered as a source code record of the design input, and would include the control structures of the run-time language as well as additional format information.

Three language alternatives existed: use CSDL as it exists, or with slight modifications; find some other language; or design a new language, either from scratch or from existing programming languages.

CSDL, or a modification of it, was considered as a first alternative in language selection, since its BNF notation has been written out completely, thus facilitating the development of a compiler or a syntax directed editor. However, since its development CSDL has not become a widely used language. While it is conceptually helpful as a model, a decision was made not to become tied to the actual syntax of CSDL, but to consider the second and third alternatives.

Concerning the second alternative, finding another language, few other control system design languages exist. While the idea of using a derived CONLAN language is appealing, CONLAN is still in the developmental stages.

Consideration was also given to developing a new language or adapting an existing language (specifically Ada) to control system design and description. Contributors to Reference 75 considered the required components of a hardware description language, and had mixed conclusions regarding adaptation of Ada as a hardware description language. However, Ada does meet many of the guidelines offered earlier in this section regarding development of hardware description and design languages. Also, Ada has been adapted to be used as a program design language, PDL/Ada, as described in References 76 and 77.

PDL/Ada was developed for three reasons:

Ada will become a standard programming language for embedded systems, and use of PDL/Ada will ease the transition from design to code;

Ada is a state of the art language, with support for software engineering concepts, and is also carefully controlled;

The Ada environment calls for many support tools which will also be able to be used with PDL/Ada.

In PDL/Ada, components of Ada were used as a basis for developing a software design language which could be processed with an Ada compiler for error checking, but not actually executed. PDL/Ada is a mapping from PDL into Ada using:

some of Ada exactly as written;

some of Ada with additional constraints on its use;

some design information written as comments in standard Ada;

some features built from Ada primitives; and

an extension of Ada's standard package to include definitions useful to program design.

PDL/Ada uses the same syntax as Ada.

For reasons similar to those which led to the development of PDL/Ada, the decision was made to develop CSD/Ada, adapting Ada to be used as a control system description, using CSDL as a guide. First, it appears that Ada will become a more widely used language as it is implemented in Department of Defense embedded systems. Second, Ada's strong typing provides a means to control and group global and local variables used in the design. Additionally, use of Ada makes it possible to develop a standard package of types available for variables to be used in designs.

Following the PDL/Ada model, a design standard package containing acceptable variable types was written for CSD/Ada. The Design Standard Package and a model for programs in CSD/Ada can be found in Appendix A. The run-time data structures, written in Pascal, can be found in the implementation code contained in Appendix C.

2. User Dialogue and Command Language

Given the design criteria that the user not have to learn a specific design/programming language, several alternatives existed regarding the design of the dialogue and command language. One was having the designer enter subroutine information using the algorithmic language, with an editor then translating the input to CSD/Ada. However, this would still require that the designer learn a language for the subroutines and a format for the remainder of the information.

A second alternative was using a syntax directed editor to lead the designer through the full CSD/Ada format, with requests for the appropriate information. This would reduce both the amount of typing required from the designer, and the opportunities afforded the designer to make errors. A syntax directed editor (SDE), discussed in References 78 and 79, and also known as a grammar driven [Ref. 80], language directed [Ref. 81], partially compiling [Ref. 82], or smart [Ref. 83] editor, allows a programmer to focus on the meaning rather than the punctuation and vocabulary when writing a program. One example is the Cornell Program Synthesizer, described in References 84 and 85, used as a tool in teaching programming and designed to allow the user to concentrate on the abstractions of a program rather than on the syntax of a specific programming language, thus supporting top-down programming.

SDEs are generally developed from the BNF notation for a language, and, with prompts, lead the programmer through a program from beginning to end. Using a menu or coded keys, the programmer types in the code for a specific component in or construct of a program. The system then shows the user the format for the construct and gives the opportunity to select from a list of possible alternatives

to fill in the blanks. This ensures that the programs written with an SDE are always syntactically correct.

While better than the first alternative, this approach was also not ideal, since the designer would still be required to learn a specific language format, if not syntax. This would be appropriate were the system being used as a teaching tool. However, in this case the purpose of using an SDE is to make it possible for the designer to avoid having to learn either the syntax or the format of any specific language.

A third alternative, use of a partial SDE, was considered and selected as the most desirable option. The SDE would lead the designer through the statement block of a subroutine and then request declaration information on variables used in the subroutine. With this partial SDE, the designer would not have to trace through the entire program structure of both declarations and statements, but could instead concentrate on the statement components (in CSD/Ada the contingency functions and task procedures), with the editor then obtaining the remaining declaration type information needed. Variable types would be limited, through menu selection, to those types included in the CSD/Ada Design Standard package. This limitation is in line with Reference 86 which points out that parsers are often more complex than needed, and can be simplified by limiting programmers to a tightly controlled list of alternatives rather than being able to handle all possible entries. Reference 82 also contains this suggestion, particularly regarding types allowed for variables.

Additionally, a means would be provided to do semantic type checking during design data entry. This approach is suggested in References 87, 88 and 89, which describe an "incremental programming environment (IPE)" of which an SDE is only part. In the IPE, the SDE and the

incremental program constructor work together interactively with the programmer. As parts of the program are entered through the SDE they are checked for semantics. Then, if the semantics are correct, intermediate code is generated, and the pieces of the program are debugged. It appeared that the semantic type checking of the IPE could be applied to the control system design environment, using the run-time data structures developed for design data entry.

Finally, lists and prompts would be used to obtain the remaining identification, criteria, and contingency/task pair information.

Three options existed as ways to have the user enter choices during data entry: type in the choice word (either all of it or part, say the first three letters), use a menu, or use 'softkeys'. The first alternative was eliminated immediately for two reasons. One, it required excessive typing from the designer. Two, it introduced related opportunities for errors. Both of these consequences could create tedious, repetitive work for the user, contrary to the design principles of the system.

With a menu, a user is required to move a marker of some sort (a box, an x, or an arrow) through a list of options on the screen, using cursor position keys, until it indicates the choice being made. When the marker is next to the choice, the enter key is pressed to indicate the selection. While not as tedious as entering whole words, several actions can be required from the user to get the marker to the desired choice in the list, and some eye-hand coordination is required to know when to stop. For that reason this option was also eliminated.

Use of softkeys [Ref. 90] also involves display of a list of choices on the screen, with each choice coded to a key on the keyboard. To indicate a choice, the user presses the key paired with it. Possible choices can be changed by

changing the screen display. The use of softkeys appeared to allow maximum flexibility of choices while minimizing both the work required from the user and the opportunity for user error. For these reasons it was selected as the primary means for data entry. However, since use of softkeys is based on a menu type display, and 'menu' is a more commonly recognized term, 'menu' will be used in the remainder of this paper to refer to the softkey means of data input.

3. Screen Display

With the dialogue and command language decisions made, a need still existed for a way to make the user aware of the choices available at each step in the design data entry process. To avoid user confusion and maintain consistency as far as where to look for input options, echo of data entered, and design written in CSD/Ada, the decision was made to use a standard screen format, with categories of information always in the same areas of the screen. Additionally, the decision was made to provide the user with a list of choices at each step in the data entry process. This would make the system self-explanatory and minimize the need for written instructions for the user, following the recommendations in Reference 52.

E. BASIC INPUT AND OUTPUT STEPS

As a result of the design decisions outlined above, it was determined that the input of design information would have five modular components:

obtain identification information through the use of prompts;

obtain the contingency/task pair data through the use of prompts and menus and link it to the related function and procedure;

obtain the criteria information through the use of prompts and menus;

use a partial syntax directed editor to obtain the statement blocks of the functions and procedures, making lists of variable names as they are used;

traverse the list of variables when the entry of each subroutine is complete, making up both the environmental table and the local variable table.

Users will be able to select the order in which they will work in these areas, with the only limitation that global and local variable lists be updated following the entry of each subroutine.

Once the user decides to exit the design entry phase, data will be passed to the second phase of the system. In phase two the user has three options: convert the data for a realization, make changes in the data, or write the data to a file in CSD/Ada format. If the user decides to convert the data to primitive list format for a realization, the system will check to ensure that data has been entered in each category and subroutines have been linked to the appropriate contingency/task data. If the decision is made to write the data to a file, identification information, contingency/task pair information, and design criteria will be written as comments in a conventional Ada package, with the environment table and the subroutines written as declarations with comments and subroutines in a conventional Ada package.

Chapter Four details the actual implementation of this system.

IV. IMPLEMENTATION

The system was implemented basically as planned and has been run successfully. An example of the system output is included as Appendix E. Implementation code is included as Appendix C.

A. DATA ORGANIZATION

Organization of run-time data into a hierarchy of records, as mentioned in Chapter Three, and shown in Figure 4.1, provided the basic structure for implementation. A header record, the root of the data tree, was used to provide pointers to identification, criteria, contingency/task pair, environment, function, and procedure records. Multiple records were able to be added in a linked list form for repeating contingency/task, environment, and subroutine occurrences, while single records were used for identification and criteria data. In some data subsections, an occurrence also became the root of a subtree of records containing specific types of data, such as comments, volumes and monitors for criteria data, or local variables and statements for subroutines. Additionally, pointers were used to link the contingency/task pair record and the function and procedure records to which it referred.

B. PROGRAM ORGANIZATION

Five major sets of subroutines were needed for data entry and manipulation. Listed from highest to lowest priority with regard to initial implementation, they were:

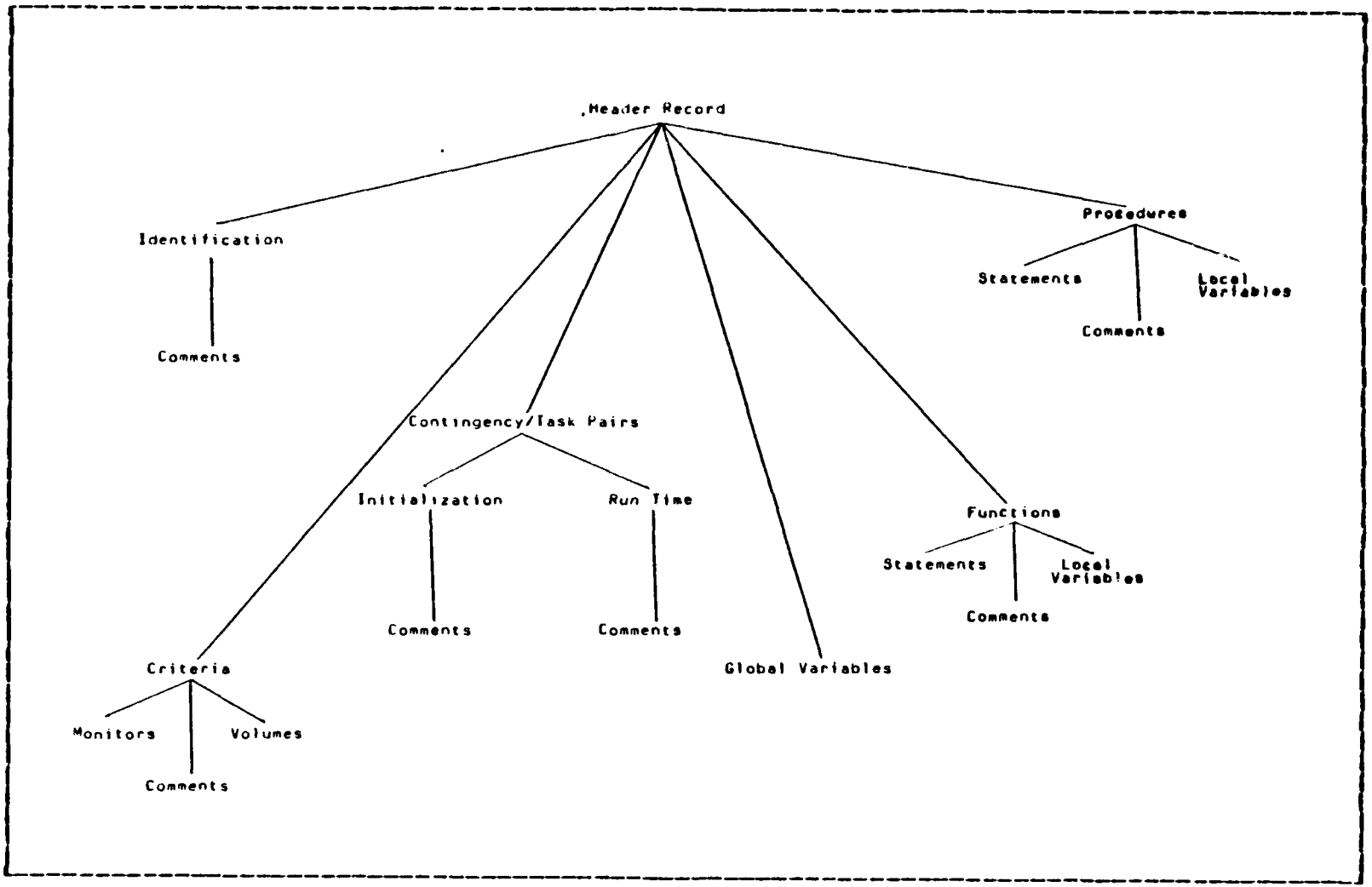


Figure 4.1 Hierarchy of Run Time Data Structures.

entry : obtain original information from designer;

write : read information from a record, write it to a permanent file for storage;

change : make changes to data already entered in records;

convert : convert data in the records to primitive list format for realization;

read : read information from a file already created back into records to work with it.

Within each set, lower level subroutines were closely matched to the record with which they would be used.

Additionally, routines were needed to handle graphics displays to the screen. This included instructions, menus, messages, and display of data entered.

Subroutines were categorized and placed in one of fourteen files, as described below. Subroutines within each file were divided into categories by level of complexity, for example, primitive routines and higher level routines which called them, and by category of design data with which they dealt, for example identification or contingency/task pair.

1. Design

DESIGN.PAS is the main file. It contains code to tell the system compiler to include all other files, and it contains the main subroutine and variables which it uses. It also contains introductory and instructional subroutines, as well as other subroutines called directly by the main subroutine or for which other appropriate files have not yet been created.

2. Records

RECORDS.PAS contains the data types and data structures created for the run-time environment. While it consists primarily of records and pointers to them, it also

contains some basic type declarations such as 'line_size' and 'name_size' used with character strings, and various enumerated types used in the records.

3. Screen

SCREEN.PAS contains the primitive graphics subroutines, for example those used to change screen set-up codes and alert the terminal to an upcoming graphics command. Additionally, it contains subroutines to draw the screen, erase the total screen or specific areas of it, and position the cursor to write a message.

4. Text

TEXT.PAS consists of subroutines to write the title screen and instructional screens.

5. Messages

MESSAGES.PAS contains one subroutine, 'Message', which uses a case statement to write various messages, called by number, to the message area on the screen.

6. Menu Files

MENUS.PAS and MENUS2.PAS contain both primitive and higher level routines to write menus to the screen menu area. Primitive routines are used to establish cursor positions for menu lines, line markers, and instruction lines. Menu use instructions are included as subroutines to be called by the two basic types of menus used - data entry or menu code entry. Menu components which will be used by more than one menu have been written as separate subroutines. Higher level menu subroutines have been categorized by the data to which they apply : identification, criteria, contingency/task pairs, subroutines, and variables. Menus for entering variable information are in MENUS2; all other

menus are located in MENUS, along with code to tell the system compiler to include the MENUS2 file.

7. Display

DISPLAY.PAS consists of subroutines used to write data entered in the run-time records to the screen for feedback to the user. A lower level subroutine is used to establish standard cursor positions for each line, with higher level routines categorized by data section.

8. Check

CHECK.PAS contains primitive routines called in both the entry of data and the changing of data to check character strings, digit strings, and traverse linked lists of records checking for matching names.

9. Change

CHANGE.PAS contains routines used the make changes in data already entered and displayed on the screen.

10. Entry Files

ENTER.PAS and ENTER2.PAS contain subroutines used for the actual data entry. ENTER contains primitive routines to enter character strings, digit strings, and lines of comments, and check subroutine names already entered, as well as higher level routines used in entry of identification and contingency/task data. It also includes the main data entry routine, and code to alert the system compiler to include ENTER2. ENTER2 consists of the subroutines used to enter criteria data, functions and procedures, and variable data.

11. Files

FILES.PAS consists of subroutines to open a file, write the data from the run-time data structures to it in a format similar to Ada, and then close the file. Subroutines are categorized by the section of data with which they are concerned, such as criteria or contingency/task pairs. Additionally, a lower level routine to write comments to the file is included, and called by several of the higher level routines.

12. Convert

CONVERT.PAS contains routines used in conversion from the run-time data structure format to primitive list format for realization.

C. SCREEN LAYOUT

Display of instructions uses the entire screen. For all other actions, the screen is divided into four areas - menu and menu instructions, input, messages, and data display - with text in each area color coordinated, as shown in Figure 4.2.

D. SYSTEM OPERATION

System operation begins with the display of a title screen. At this point the user is given the option of reading some general instructions first or beginning data entry immediately.

Design data entry follows a path from the more general areas to the more specific. The user is allowed to pick one of five areas in which to enter data - identification, criteria, contingency/task pairs, functions, or procedures, or to decide to exit the data entry section. Once one of

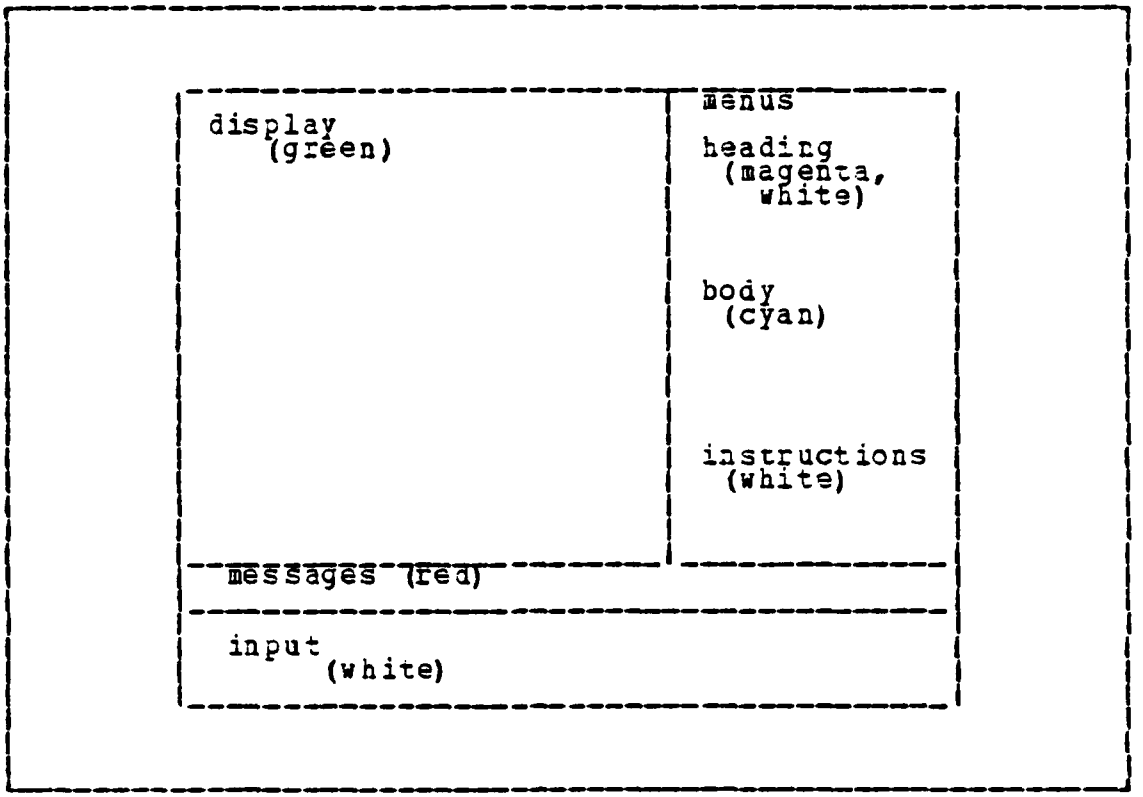


Figure 4.2 Screen Layout.

the data areas is selected, the user is led through all of the data items needed for one set of data in that area, with menus used to prompt and limit the data choices which can be entered. Entries are echoed immediately in the input area on the screen. In addition, when all data has been entered in a section, the complete set of data is written to the screen in the display area. Although not yet fully implemented, at this point the designer will, in the future, be able to make changes in the data displayed.

Following data entry for a section, the user is returned to the start of the data entry loop and again allowed to pick one of the five data entry areas or exit, with the condition that data can be entered in the identification and criteria areas only once.

When the user is done entering data and decides to exit, the system moves to phase two, data manipulation. Although not yet fully implemented, three options are available. Changes can be made to the data, it can be converted to a primitive list format, and a realization can be generated, or it can be writtten to a permanent file. Again, the user continues returning to these options in a loop until he or she decides to exit from phase two. The exit from phase two terminates the program.

E. ERROR CHECKING

There are several components of the error checking system. In general, data is checked when it is entered. When an error is found by one of the components, or confirmation of an entry is needed, a message is written to the message area of the screen with specific information.

1. Menu Entry Checks

Entry of data takes one of three forms. The user can be asked to enter either the code for one of the chcices on the menu, a character string, or a numerical value. In the first case, if the user enters a code not in the menu, a message will be written explaining the error and asking that the entry be repeated. In the second and third cases, specific subroutines are used to check data entered.

a. Check_Item

'Check_Item' is called when the user is entering identifiers, and it checks for errors commonly made in variable and subroutine names, as well as specific design system constraints. It will give an error message and ask for a repeat of the entry if the identifier is longer than ten characters, does not start with a letter, or contains

blanks. Entry of only a carriage return will result in a string of length zero, and the user will be asked to repeat the process.

b. Check_Digits

'Check_Digits' is called when the user is entering numerical values. It reads input as a character string and checks to ensure that length does not exceed six digits (a size selected to ensure that a set of contingency/task data can be writtten as one line of a file), and only digits have been entered. It then calls a system library routine to convert the character string of digits to its integer value. Entry of only a carriage return will result in a value of zero being assigned to the variable in question.

2. Subroutine Name Checking

Both contingency/task and subroutine data entry routines do extensive subroutine name checking, based on the premise that each function and procedure will be part of only one contingency/task pair. Subroutine records can be entered through either subroutine entry procedures or contingency/task pair entry procedures. Both will enter name and subroutine kind. However, subroutine entry will enter statements and set the contingency/task pointer to nil. Contingency/task pair entry will enter a contingency/task pointer and set the statement pointer to nil. This makes it possible to determine how much of the data related to a subroutine has been entered.

When entering contingency/task pair data, a check will be made of both the initialization and run-time lists to ensure that neither the function or subroutine has been included in a contingency/task pair already entered. A check will then be made of the function and procedure lists.

If the subroutine has already been entered, it will be linked to the contingency/task pair record. If not, a new subroutine record will be added, the name and kind entered with the other fields set to nil, and the subroutine record will be linked to a contingency/task pair record.

When a subroutine is to be entered, a check will be made to see if the name has already been used. If it has, a check will be made of the statement pointer. If it is not nil, a message will be returned that the entire subroutine has already been entered. If it is nil, this record will be selected as the one in which to enter the rest of the subroutine data. If the name is not located, a new record will be added, the contingency/task pointer set to nil, and the rest of the subroutine data entered.

3. Statement Component Checking

Use of the partial syntax directed editor for entry of statements ensures that all statements entered are syntactically correct. Menus used to list options for information entered about variables, such as precision or technology, minimize the chance of entry of incorrect data by the user. Finally, use of pointers to link all variables in an expression makes it possible to implement more detailed type checking as work on the system continues in the future.

4. Completion Check

When the user decides to convert the data to primitive list format, a check will be made to ensure that all data needed has been entered. The first check will be of the header record, to see that identification and criteria links are not nil. Then, based on the procedure for subroutine record entry described above, a check will be made of the function and procedure lists, to make sure that

each subroutine has both statements entered and a pointer linking it to a contingency/task pair record.

5. File Notation

As a reminder to the designer, and for future use when data in files can be read back into the system, 'no data entered' is written in sections of the file for which no data at all has been entered.

6. Exit Check

Since exiting from phase two of the system will result in destruction of the run-time data structure, a two step procedure is used to confirm that the user does actually want to terminate the design process.

Chapter Five will evaluate the system, discuss problems encountered in implementing it, and suggest areas for future work.

V. CONCLUSIONS AND RECOMMENDATIONS

A. GOALS OF THE PROJECT

The goals originally established with regard to this project were to design a workstation which would not require its user to learn a specific language for data entry, present a user-friendly approach to the user, and produce an output of the data entered in a user readable format which could also be converted to a primitive list format. Considering this project as a first implementation, these goals have been met. Use of the system has shown that, for a user familiar with the general concepts of Matelan's Computer System Design Language and Ross' computer aided design process, the screen instruction and error checking in the system are sufficient to ensure correct and complete data entry. The preliminary work done in making changes to data already entered indicates that it should be possible to expand the system to read in data from a file and repeatedly modify data for different realizations.

B. PROBLEM AREAS

While the system was implemented successfully, there were difficulties in working with the GIGI terminal. Although GIGI's graphics language can be embedded within Pascal, there was little documentation available on how to use it in an interactive setting and how to use it other than to draw pictures. The most significant problem occurred in code sequences which shifted the terminal from an interactive Pascal command mode, such as reading an input, to a graphics command mode, such as writing a message. A '!' at the start of a sequence of code is the

signal to indicate to the terminal that the code following it is a graphics display instruction. While the '!' at the start of a graphics command is sufficient to alert the terminal within a sequence of graphics actions, it is not sufficient to switch the terminal from Pascal to graphics. It was necessary to include an additional wake up '!', included as the first command in graphics routines, such as clearing areas of the screen, likely to be called immediately following Pascal commands.

An additional problem occurred in writing data entered in records back to the screen for display. While it was not difficult to determine how to write a specific character string to the screen, it was less obvious how to write the value of a variable or contents of a record field to the screen. The necessary procedure is less than logical, and was found through trial and error rather than in available documentation.

Finally, writing the graphics commands for each line of text was found to be tedious. Use of the GIGI macros was considered but decided against for two reasons. When first used the macros were found to be unreliable, although this may have been due to the problems of switching from Pascal to graphics commands mentioned earlier. More importantly, the use of macros appeared to decrease, rather than increase, readability. This is because they can be identified by only one letter, rather than by a word that indicates function. An attempt was made to replace the graphics command strings with Pascal variables of type 'packed array of characters' which had been assigned the graphics command strings as values. However, this attempt to make the graphics commands more readable and reduce required typing was unsuccessful.

C. FUTURE WORK

There are several areas for future work as a result of this project. In the field of computer aided design system design there is much room for expansion of the current system. The syntax directed editor can be enlarged to allow for a full block structured language. The contingency/task data options not yet implemented can be added. The means for data change and conversion to primitive list format can be fully implemented. Finally, the means to read data in from a previously written file can be designed and implemented.

With regard to writing of data to a file, there is room for system modification and further expansion. The use of a standard file name and file type is based on use of an operating system which numbers file copies, and retains previous copies until the user deletes them, rather than writing over them when an additional file of the same name and type is created. It would be beneficial to allow the user to enter the file name and file type when the decision is made to write the data to a file. This will allow different versions of a design to have different names. Additionally, there is work to be done in strengthening the link between CSD/Ada and Ada. Included in this is adding to the subroutines used to write the run-time data to a file so that the output more closely matches Ada format.

In the field of user-friendliness, there is opportunity to expand and add flexibilty to the instructions included within the design system. Introductory instructions can be increased, and the user can be given the opportunity to select specific instruction screens and enter and exit the instructions at other than the beginning and end of the sequence of screens. It would also be worthwhile to take advantage of the use of case statements for menu code entry

and add a code for the option of a help request. This would make information in a specific area available during the data entry process.

Much work remains to be done in expanding and improving the system. However, it is clear that this project has shown that it is possible to design and implement a user-friendly work station for computer aided design.

LIST OF REFERENCES

List of Abbreviations Used:

ACM	Association for Computing Machinery
AFIPS	American Federation of Information-Processing Societies
IEE	Institution of Electrical Engineers
IEEE	Institute of Electrical and Electronics Engineers

1. Treu, S., "A Testbed for Providing Uniformity to User-Computer Interaction Languages," National Bureau Of Standards, 1980.

2. Ivie, E. L., "The Programmer's Workbench - A Machine for Software Development," Communications of the ACM, Vol. 20 #10, October 1977, pp. 746-753.

3. O'Neill, L. A., Savolaine, C. G., Thompson, T. J., Franke, J. M., Friedenson, R. A., Walsh, E. D., McDonald, P. B., Breiland, J. R., Evans, D. S., "Designers Workbench - Efficient and Economical Design Aids," 16th Design Automation Conference Proceedings, pp. 185-195, IEEE, 1979.

4. McWilliams, T. E., and Widdoes, L. C. Jr., "SCALD: Structured Computer-Aided Logic Design," 15th Design Automation Conference Proceedings, pp. 271-277, IEEE, 1978.

5. McWilliams, T. E., and Widdoes, L. C. Jr., "The SCALD Physical Design Subsystem," 15th Design Automation Conference Proceedings, pp. 278-284, IEEE, 1978.

6. Barbacci, M. R., and Siewiorek, D. P., "Application of an ISP Compiler in a Design Automation Laboratory," International Symposium on Computer Hardware Description Languages, pp. 69-75, IEEE, 1975.

7. Hafer, L. J., and Parker, A. C., "Register-Transfer Level Digital Design Automation: The Design Process," 15th Design Automation Conference Proceedings, pp. 213-219, IEEE, 1978.

8. Snow, E. A., Siewiorek, D. P., Thomas, D. E., "A Technology-Relative Computer-Aided Design System: Abstract Representations, Transformations, and Design Tradeoffs," 15th Design Automation Conference Proceedings, pp. 220-226, IEEE, 1978.

9. Barbacci, M. R., "ISP Specifications," 16th Design Automation Conference Proceedings, pp. 64-72, IEEE, 1979.

10. Parker, A. C., Thomas, D., Siewiorek, D., Barbacci, M., Hafer, L., Leive, G., Kim, J., "The CMU Design Automation System," 16th Design Automation Conference Proceedings, pp. 73-80, IEEE, 1979.

11. Ross, A. A., Computer Aided Design of Microprocessor-Based Controllers, Ph. D. Dissertation, University of California, Davis, June 1978.

12. Ross, A. A., and Loomis, H. H. Jr., "Computer Aided Design of Microprocessor-Based Systems," 15th Design Automation Conference Proceedings, pp. 227-230, IEEE, 1978.

13. Biehl, G., and Dietzinger, A., "Computer-Aided Design of Microprocessor-Based Digital Controllers," Microprocessors and Microprogramming, Vol. 7 #5, May 1981, pp. 326-333, The Netherlands.

14. Rader, J. A., "Evolution of the Philosophy and Capability for the CAD of Digital Modules," 10th Design Automation Workshop Proceedings, pp. 282-288, IEEE, 1973.

15. Armstrong, R. A., "A CAD User's Perspective: What Gets Done Right, Wrong, and Not at All," 17th Design Automation Conference Proceedings, p. 517, IEEE, 1980.

16. Damodaran, L., and Eason, K. D., "Design Procedures for User Involvement and User Support," pp. 373-388, Coombs, M. J., and Alty, J. L., Computing Skills and the User Interface, Academic Press, 1981.

17. Peled, J., and Carroll, M. P., "The 'Gap' Between Users and Designers of CAD/CAM Systems: Search for Solutions," 18th Design Automation Conference Proceedings, pp. 703-705, IEEE, 1981.

18. Chu, Y., "Why Do We Need CHDL's?," Computer, Vol. 7 #12, December 1974, pp. 18-19, IEEE Computer Society.

19. Van Cleemput, W. M., "Computer Hardware Description Languages and Their Applications," 16th Design Automation Conference Proceedings, pp. 554-560, IEEE, 1979.

20. Van Cleemput, W. M., "An Hierarchical Language for the Structural Description of Digital Systems," 14th Design Automation Conference Proceedings, pp. 377-85, IEEE, 1977.

21. Chu, Y., "An Algol-Like Computer Design Language," Communications of the ACM, Vol. 8 #10, October 1965, pp. 607-615.

22. Chu, Y., Computer Organization and Microprogramming, Chapter 1, Prentice-Hall, 1972.

23. Dasgupta, S., "Computer Design and Description Languages," Advances in Computers Vol. 21, pp. 91-154, Academic Press, 1982.

24. Chu, Y., "Computer System Design Description," 19th Design Automation Conference Proceedings, pp. 842-850, IEEE, 1982.

25. Hill, F. J. and Peterson, G. R., Introduction to Switching Theory and Logical Design, Chapter 15, John Wiley & Sons, 1974.

26. Hill, F. J., "Introducing AHPL," Computer, Vol. 7 #12, December 1974, pp. 28-30, IEEE Computer Society.

27. Hill, F. J., "Updating AHPL," International Symposium on Computer Hardware Description Languages, Proceedings, pp. 22-29, IEEE, 1975.

28. Hill, F. J., and Peterson, J. R., Digital Systems : Hardware Organization and Design, Chapter 5, John Wiley and Sons, 1978.

29. Peterson, G. R., D'Souza, C., and Hill, F. J., "AHPL: A Language for Function Level Design," First Annual Phoenix Conference on Computers and Communications, Proceedings, pp. 48-53, IEEE, 1982.

30. Duley, J. R., and Dietmeyer, D. L., "A Digital System Design Language (DDL)," Transactions on Computers, Vol. C-17, September 1968, pp. 850-861, IEEE Computer Society.

31. Duley, J. R., and Dietmeyer, D. L., "Translation of a DDL Digital System Specification to Boolean Equations," Transactions on Computers, Vol. C-18, April 1969, pp. 305-313, IEEE Computer Society.

32. Dietmeyer, D. L., "Introducing DDL," Computer, Vol. 7 #2, December 1974, pp. 34-38, IEEE Computer Society.

33. Shiva, S. G., and Covington, J. A., "Modular Description/Simulation/Synthesis Using DDL," 19th Design Automation Conference Proceedings, pp. 321-329, IEEE, 1982.

34. Colladc, M., and Talavera, J. A., "Design Automation of Microprocessors," Applications of Mini and Microcomputers, pp. 27-32, Industrial Electronics and Control Instrumentation Society, 1980.

35. Bell, C. G., and Newell, A., "The PMS and ISP Descriptive Systems for Computer Structures," Spring Joint Computer Conference Proceedings, Vol. 36, pp. 351-374, AFIPS, 1970.

36. Bell, C. G., and Newell, A., Computer Structures: Readings and Examples, Chapters 1 & 2, McGraw-Hill, 1971.

37. Bartacci, M. R., "Comparison of Register Transfer Languages for Describing Computers and Digital Systems," Transactions on Computers, Vol. C-24 #2, February 1975, pp. 137-150, IEEE Computer Society.

38. Parker, A. C., Thomas, D. E., Crocker, S., Cattell, R. G. G., "ISPS: A Retrospective View," 4th International Symposium on Computer Hardware Description Languages, Proceedings, pp. 21-27, IEEE, 1979.

39. Siewiorek, D. P., Bell, C. G., Newell, A., Computer Structures: Principles and Examples, Chapters 2 and 4, McGraw-Hill, 1982.

40. Pilcty, R., Bartacci, M. R., Borrione, D., Dietmeyer, D. L., Hill, F. J., and Skelly, P., "CONLAN - A Formal Construction Method for Hardware Description Languages: Basic Principles," National Computer Conference Proceedings, Vol. 49, pp. 209-217, AFIPS, 1980.

41. Pilcty, R., Bartacci, M. R., Borrione, D., Dietmeyer, D. L., Hill, F. J., and Skelly, P., "CONLAN - A Formal Construction Method for Hardware Description Languages: Language Derivation," National Computer Conference Proceedings, Vol. 49, pp. 219-227, AFIPS, 1980.

42. Pilcty, R., Bartacci, M. R., Borrione, D., Dietmeyer, D. L., Hill, F. J., and Skelly, P., "CONLAN - A Formal Construction Method for Hardware Description Languages: Language Application," National Computer Conference Proceedings, Vol. 49, pp. 229-236, AFIPS, 1980.

43. Pilcty, R., and Borrione, D., "The CONLAN Project: Status and Future Plans," 19th Design Automation Conference Proceedings, pp. 202-212, IEEE, 1982.

44. Hill, F. J., A Summary Discussion of CONLAN, Engineering Experiment Station, College of Engineering, University of Arizona, Tucson, July 1982.

45. Matelan, M. N., Automating the Design on Dedicated Real Time Control Systems, UCRL-78651, Lawrence Livermore Laboratory, 21 August 1976.

46. Chapanis, A., Man-Machine Engineering, Wadsworth Publishing Co., Monterey, Ca., 1965.

47. Spence, R., "Human Factors in Interactive Graphics," Computer Aided Design, Vol. 8 #1, January 1976, pp. 49-53, IPC Science and Technology Press, Ltd.

48. Martin, J., Design of Man-Computer Dialogues, Prentice-Hall, 1973.

49. Black, J. L., "A General Purpose Dialogue Processor," National Computer Conference Proceedings, Vol. 46, pp. 397-408, AFIPS, 1977.

50. Friesen, O. D., "The Dialogue System: A Tool for Testing and Implementing End User Interfaces," First Annual Phoenix Conference on Computers and Communications, Proceedings, pp. 152-156, IEEE, 1982.

51. Miller, L. A., and Thomas, J. C. Jr., "Behavioral Issuers in the Use of Interactive Systems," International Journal of Man-Machine Studies, Vol. 9 #5, September, 1977, pp. 509-536, Academic Press.

52. Smith, L. B., "The Use of Interactive Graphics to Solve Numerical Problems," Communications of the ACM, Vol. 13 #10, October 1970, pp. 625-634.

53. Irby, C., Bergsteinsson, L., Moran, T., Newman, W., Tesler, T., A Methodology for User Interface Design, Xerox Palo Alto Research Center, Information Technology Group, Systems Development Division, 1977.

54. Newman, W. M. and Sproul, R. F., Principles of Interactive Computer Graphics, chapter 28, McGraw-Hill, 1979.

55. Eason, K. D., and Damodaran, L., "The Needs of the Commercial User," pp. 115-139, Coombs, M. J., and Alty, J. L., Computing Skills and the User Interface, Academic Press, 1981.

56. Foley, J. D. and Wallace, V. L., "The Art of Natural Graphic Man-Machine Conversation," Proceedings, Vol. 62 #24, April 1974, PP. 462-471, IEEE.

57. Miller, G. A., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," Psychological Review, Vol. 63, #2, pp. 81-97, American Psychological Association, 1956.

58. Haber, R. N., and L. Wilkinson, "Perceptual Components of Computer Displays," Computer Graphics and Applications, Vol. 2 #3, May 1982, pp. 23-35, IEEE Computer Society.

59. Miller, R. B., "Response Time in Man-Computer Conversational Transactions," Fall Joint Computer Conference Proceedings, Vol. 33, Pt. 1, pp. 267-277, AFIPS, 1968.

60. Spence, R. and Apperley, M., "Interactive-Graphic Man-Computer Dialogue in Computer-Aided Circuit Design," Transactions on Circuits and Systems, Vol. CAS-24, February 1977, pp. 49-61, IEEE Circuits and Systems Society.

61. Shneiderman, B., "Human Factors Experiments in Designing Interactive Systems," Computer, Vol. 12 #12, December 1979, pp. 9-19, IEEE Computer Society.

62. Mczelcc, H., "A Human/Computer Interface to Accomodate Learning Stages," Communications of the ACM, Vol. 25 #2, February 1982, pp. 100-104.

63. Walther, G. H., and O'Neill, H. F. Jr., "On Line User-Computer Interface - The Effects of Interface Flexibility, Terminal Type, and Experience on Performance," National Computer Conference Proceedings, Vol. 43, pp. 379-384, AFIPS, 1974.

64. Relles, N. and Price, L., "A User Interface for Online Assistance," 5th International Conference on Software Engineering, pp. 400-408, IEEE, 1981.

65. Price, L., "Design of Command Menus for CAD Systems," 19th Design Automation Conference Proceedings, pp. 453-459, IEEE, 1982.

66. Irby, C. H., "Display Techniques for Interactive Text Manipulation," National Computer Conference Proceedings, Vol. 43, pp. 247-255, AFIPS, 1980.

67. Shneiderman, B., Software Psychology: Human Factors in Computer and Information Systems, Cambridge, Massachusetts, Winthrop Publishers, 1980.

68. Thomas, D. E., "The Automatic Synthesis of Digital Systems," Proceedings, Vol. 69 #10, October 1981, pp. 1200-1211, IEEE.

69. GIGI/ReGIS Handbook, Digital Equipment Corporation, Maynard, Massachusetts, June, 1981.

70. Stevens, W. P., Myers, G. J., Constantine, L. L., "Structured Design," IBM Systems Journal, Vol. 13 #2, 1974.

71. Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, Vol. 15, #12, December 1972, pp. 1053-1058.

72. Parnas, D. L., "Designing Software for Ease of Extension and Contraction," Transactions on Software Engineering, Vol. SE-5 #2, March 1979, pp. 128-137, IEEE.

73. Miller, T. J., and Vellenga, J. H., "A High Level Language for VLSI Design," First Annual Phoenix Conference on Computers and Communications, pp. 41-44, IEEE, 1982.

74. Manwaring, M. L., "A Computer Aided Approach to the Design of Digital System Controllers Using Microprocessors," 12th Asilomar Conference on Circuits, Systems, and Computers, pp. 431-436, IEEE, 1979.

75. Preston, C. W., Report of IDA Summer Study on Hardware Description Language, Institute for Defense Analysis, Science and Technology Division, IDA Paper P-1595, October 1981.

76. Waugh, D. W., "Ada as a Design Language," IBM Software Engineering Exchange, October 1980, pp. 8-12.

77. Schwartz, L. "PDL/ADA: A Design Language, A Transition Tool," IBM Federal Systems Division, Lecture, Naval Postgraduate School, Monterey, 3 February 1983.

78. MacLennan, B. J., The Automatic Generation of Syntax Directed Editors, Technical Report, Naval Postgraduate School, October 1981.

79. Computer Aided Design/Software User Manuals: Syntax Directed Editor (SDE), System Development Corporation, TM-7812/000/00, August 1982.

80. Shockley, W. R., and Haddow, D. P., A Conceptual Framework for Grammar-Driven Synthesis, Master's Thesis, Naval Postgraduate School, December 1980.

81. Morris, J. M., and Schwartz, M. D., "The Design of a Language-Directed Editor for Block Structured Languages," SIGPLAN Notices, Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Vol. 16 #6, June 1981, pp. 28-33, Association for Computing Machines.

82. Connell, J. M., and Munsil, W. E., "Considerations for a Partially Compiling Editor," First Annual Phoenix Conference on Computers and Communications, pp. 121-124, IEEE, 1982.

83. Denning, P. J., "Smart Editors," Communications of the ACM, Vol. 24 #8, August 1981, pp. 491-493.

84. Teitelbaum, T., "The Cornell Program Synthesizer: A Syntax Directed Programming Environment," SIGPLAN Notices, Vol. 14 #10, October 1979, p. 75, Association For Computing Machines.

85. Teitelbaum, T., Reps, T., and Horowitz, S., "The Why and Wherefore of the Cornell Program Synthesizer," SIGPLAN Notices, Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Vol. 16 #6, June 1981, pp. 8-16, Association For Computing Machines.

86. Shani, U., "An Approach to Graceful Man-Machine Communication," First Annual Phoenix Conference on Computers and Communications, pp. 148-151, IEEE, 1982.

87. Notkin, D. S., and Haberman, A. N., Software Development Environment Issues as Related to Ada, Department of Computer Science, Technical Report, Carnegie-Mellon University, 1979.

88. Medina-Mora, R., and Feiler, P. H., "An Incremental Programming Environment," Transactions on Software Engineering, Vol. SE-7 #6, September 1981, pp. 472-482, IEEE.

89. Medina-Mora, R., Syntax-Directed Editing: Towards Integrated Programming Environments, Ph. D. Dissertation, Carnegie-Mellon University, Pittsburgh, Pennsylvania, March, 1982.

90. Sarcnas, T., "Improving the Man/Machine Interface," New Electronics, Vol. 12 #19, pp. 86, 89-90, Great Britain, 2 October 1979.

APPENDIX A

CSD/ADA

```
Package design_standard is

    type analog is integer;
    type bocln is boclean;  -- * true/false
    type ext_fixed is integer;
    type ext_float is integer;
    type int_fixed is integer;
    type int_float is integer;

end design_standard;

with design_standard; use design_standard;

Package control_system_design is

    -- identification block

        -- identification data

    -- criteria block

        -- criteria data

    -- contingency/task block

        -- contingency/task pairs data

    -- environment table

        variable : type; -- additional global variable data

    -- subroutine list

        function name (input) returns type;

        procedure name (input; output);

end control_system_design;


package body control_system_design is

    (* complete code for functions and procedures
                listed here *)

end control_system_design;
```

```
--IDENTIFICATION DATA

-- Name:      barbara J. Sherlock
-- Project:   Output Demonstration
-- Date:      7 June 1983

-- Demonstration of system output for inclusion in thesis



-- CRITERIA DATA

-- Criteria:   FIRST
-- fully implemented
-- monitors:    1   2   3   4   5
-- volumes:     9   8   7   6   5   4   3   2   1   0



-- CONTINGENCY/TASK PAIR DATA

-- CONTINGENCY      TASK        DEF     RHO   UNITS   BETA1   BETA2  GLOBAL  LOCAL  GAMMA1  GAMMA2   BGD    CONDITION

-- Initialization

-- functone      proctone      WHEN     200    MS      100     100      1       1       0       0     TRUE     NONE
-- Implementation
--    limitations -
--          contingency definition - only 'when' implemented
--          condition - only 'none' implemented
--    change implemented for numerical value data (rho, gamma1, etc.) entered

-- Run Time

-- functwo       proctwo       WHEN     100    MS      100     100      1       1       0       0     FALSE    NONE



-- VARIABLES

a          : EXT-FLOAT;
--        value      addr       prec        min        max       mantis     expon       port
--          0        999         16          0        512          6         10          1



-- FUNCTIONS

function functone  ;
```

```
-- VARIABLES

-- no variables entered


BEGIN
 ISSUE(a          );
END;


function func+two  ;


-- VARIABLES

-- no variables entered


BEGIN
 SENSE(a          );
END;




-- PROCEDURES

procedure proc+one  ;
-- implementation -
--       variables - only external floating point type implemented
--       statements - only input/output statements inplemented


-- VARIABLES

b          : EXT+FLOAT;
--         value      addr       prec       min       max      mantis     expon      port
--           0          1         32         0       32000       20         10        999


BEGIN
 SENSE(a          );
 ISSUE(b          );
END;


procedure proc+two  ;


-- VARIABLES

a          : EXT+FLOAT;
--         value      addr       prec       min       max      mantis     expon      port
--           0        9999        64         0        999       32         32         99


BEGIN
```

```
 ISSUE(a          );
END;
```

```
{file design.pas - main file}

[INHERIT ('SYS$LIBRARY:STARLET')]

program design (input,output,data);

{main program - will call all other files
using % include}

%include 'records.pas'
{include record types for run time data}

   var
        done_main : boolean;
        choice : char;
        exit_choice : char;

        design_data : design_ptr;

        data : text;  {to open file}

%include 'screen.pas/nolist'

%include 'text.pas/nolist'

%include 'messages.pas/nolist'

%include 'menus.pas/nolist'

%include 'display.pas/nolist'

%include 'check.pas/nolist'

%include 'change.pas/nolist'

%include 'enter.pas/nolist'

{%include 'convert.pas/nolist'}

%include 'files.pas/nolist'



PROCEDURE continue;
        {will read input until user enters a "y" - used
        to wait for signal to go on to next screen
        of instructions}
   var
        done_cont : boolean;
        choice : char;
begin
        done_cont := false;
        repeat  {until done_cont true}
          readln (choice);
```

```
          clear*msg;
          case choice of
                'y':
                    done*cont := true;
               otherwise
                    message (10);  {enter 'y'}
          end;  {case}
        until done*cont;
end;  {continue}



PROCEDURE instructions;
        {will call each instruction screen and await a signal from the
        user to proceed}
    var
        choice : char;

begin
        instruct*text*1;
        continue;
        instruct*text*2;
        continue;
        instruct*text*3;
        continue;
end; {instructions}


PROCEDURE write*msg (m*num:integer);
        {will write message and erase when acknowledged}
        {may not be used}
        {may be useful when help messages available for
        error message called}

        var
                choice : char;


begin
        message (m*num);
        repeat {until} length(choice) = 0)
                readln (choice);
        until length(choice) = 0;
        clear*msg;
end; {write*msg}



PROCEDURE intro;
        {will call opening screen and then, if user requests it, design
        instructions}

    var
        choice : char;
        done*intro : boolean;
begin
        done*intro := false;
        title*text;
        repeat  {until done*intro}
          readln (choice);
```

```
            clear_msg;
            case choice of
               'n':
                  done_intro := true;
               'y':
                  begin
                    instructions;
                    done_intro := true;
                  end;
               otherwise
                  message (9);  {enter 'y' or 'n'}
            end;  {case}
         until done_intro;

end; {intro}


PROCEDURE confirm (m_num:integer; var choice:char);

         {will confirm user wishes to terminate design and
         destroy run time data structures}

begin
         clear_msg;
         repeat {until choice = y or n}
                  message (4);
                  readln (choice);
                  clear_msg;
         until ((choice = 'y') or (choice = 'n'));
end; {confirm}



PROCEDURE convert_main;

begin
         message_posit;
         writeln ('!t(5],w(v,i(r)))''convert main called');
end; {convert_main}



PROCEDURE main;

begin
         init_init;
         intro;
         enter_new_main (design_data);
         done_main := false;
         repeat {until done_main = true}
                  clear_input;
```

```
                phase_two_menu;
                repeat (choice);
                clear_msg;
                case choice of
                        '1':    change_main (design_data);
                        '2':    convert_main;
                        '3':    file_main (design_data);
                        '4':    {exit}
                                begin
                                        confirm (4,exit_choice);
                                        if exit_choice = 'y' then
                                         done_main := true;
                                end;
                        otherwise

                           begin
                                message (1);
                                clear_inout;
                           end;
                end; {case}

        until done_main;

end; {main}

%include 'test.pas'

begin

        gial_init;

        test;

        reset_scroll;

end. {design}
```

```
{file records.pas - record type declarations}

{run time data structures for design}


type


{preliminary types}

line_size = varying[80] of char;

name_size = packed array [1..10] of char;

digits = varying [10] of char;

precis1 = 8..64;
precis2 = 16..64;
units = (h,m,s,ms,us,ns);
tech = (ttl, ecl, i2l);
internal_type = (vbl, temp, cons);
ct_kind = (init, run);
subrout_kind = (func, proc);
crit_option = (first, cost, power);

scope_types = (global, local);


{preliminary records}

comt_ptr = ^comment;
comment = record
        line : line_size;
        next_comt : comt_ptr;
end; {comment}



taql = (analog,booln,ext_fixed,ext_float,int_fixed,int_float);

vbl_ptr = ^variable;
variable = record

        name : name_size;
        value : integer;
        mem_addr : integer;
        next_vbl : vbl_ptr;

        case kind1 : taql of

           analog: (precision_a : precis1;
                min_val_a : integer;
                max_val_a : integer;
                sample_rate_time_a : integer;
                sample_rate_units_a : units);

           booln: (technology_b : tech;
```

```
                bool*value : boolean;
                port*num* : integer));

            ext*fixed: (precision*fx : precis1;
                min*val*fx : integer;
                max  il*fx : integer;
                technology*fx : tech;
                port*num*fx : integer);

            ext*float: (precision*fl : precis2;
                min*val*fl : integer;
                max*val*fl : integer;
                technology*fl : tech;
                mantis*fl : integer;
                expon*fl : integer;
                port*num*fl : integer);

            int*fixed:
                (kind2 : internal*type;
                    precision*ifx :  precis1;
                    min*val*ifx : integer;
                    max*val*ifx : integer;

                    cons*value*ifx : integer);

            int*float:
                (kind3 : internal*type;
                     precision*ifl : precis2;
                     min*val*ifl : integer;
                     max*val*ifl : integer;
                     mantis*ifl : integer;
                     expon*ifl : integer;

                    cons*value*ifl : integer));
end; (variable record)



volumes*ptr = ↑volumes;
volumes = record
        v1,v2,v3,v4,v5,v6,v7,v8,v9,v10 : integer;

end; (volumes)


monitors*ptr = ↑monitors;
monitors = record
        m1,m2,m3,m4,m5 : integer;

end; (monitors)



operators = (add, sub, mult, divd, data*in, data*out);
tag3 = (term, nonterm);
expr*item*ptr = ↑expr*item;
expr*item = record
        case item : tag3 of
          term : (name : name*size;
```

```
                next*vbl : expr*item*ptr;
                prev*vbl : expr*item*ptr;
                expr*set : expr*item*ptr));

            nonterm : (operator : operators;
                left*ptr : expr*item*ptr;
                  {left ptr nil for unary and i/o ops}
                right*ptr : expr*item*ptr));
end; {expr*item}



relations = (eq, lt, lte, gt, gte, ne);
tag4 = (rel, log*val, log*op, vrbl, funcl);
cond*ptr = ^condition;
condition = record
        case bool*expr : tag4 of
          rel : (relation : relations;
                left*side : expr*item*ptr;
                right*side : expr*item*ptr);

          log*val : (value : boolean);

          {others not implemented yet}
           log*op, vrbl, funcl : ());
end; {condition}



tag5 = (io, assig, if*st, while*st, for*st,exit*st,perf*st);
stmt*ptr = ^statement;
statement = record
        next*stmt : stmt*ptr;
        case stmt*type : tag5 of
                io : (direction : expr*item*ptr);

                assig : (left*side : expr*item*ptr;
                        right*sidt : expr*item*ptr);

                if*st : (cond : cond*ptr;
                        then*stmt : stmt*ptr;
                        else*stmt : stmt*ptr);

                {others not implemented yet}
                while*st, for*st,exit*st,perf*st : ());
end;


{section records}

ct*pair*ptr = fct*pair;          {record forward}


subrout*ptr = ^subroutine;
subroutine = record
        kind : subrout*kind;
        name : name*size;
```

```
        ctepair : ctepairptr;
        comtelink : comtptr;
        firstvbl : expritemptr;
        locvbltlink : vbleptr;
        stmtslink : stmtptr;
        next : subroutptr;
end; {subroutine}


criteriaptr = fcriteria;
criteria = record
        crit : critoption;
        comtelink : comtptr;
        volumeslink : volumesptr;
        monitorslink : monitorsptr;
end; {criteria}


tagb = (unless, iff, none);
ctepair = record
        kind : ctekind;
        comtelink : comtptr;
        func : subroutptr;
        proc : subroutptr;
        timeunits : units;
        contdef : (when, every, at, doo);
        rno : integer;
        beta1 : integer;
        beta2 : integer;
        gloperd : integer;
        locpri : integer;
        gamma1 : integer;
        gamma2 : integer;
        background : boolean;
        nextpair : ctepairptr;
        case ctcond : tagb of
                unless, iff : (condelink : condptr);
                none : ();
end; {ctepair}


ctetypesptr = fctetypes;
ctetypes = record
        initlist : ctepairptr;
        runtlist : ctepairptr;
end;  {ctetypes}


idptr = fid;
id = record
        destoner : linesize;
        project : linesize;
        date : linesize;
        comtelink : comtptr;
end; {id}
```

```
{header record}

designptr = ↑design;
design = record
        idelink : ideptr;
        crelink : cretypesptr;
        critelink : criteriaptr;
        envelink : vbleptr;
        funcelink : subrouteptr;
        procelink : subrouteptr;
end; {design}

{end file records}
```

```
{file screen.dbs - primitive graphics subroutines}



PROCEDURE set+up+gigi;
        {changes default set ups - sets so gigi will
        read screen commands and screen will not scroll}

begin
        writeln ('',CHR(27),'PrGP1',CHR(27),'\');
        writeln ('',CHR(27),'PrSM0',CHR(27),'\');
end; {set+up+gigi}

PROCEDURE reset+scroll;
        {will reset screen to scroll}
begin
        writeln ('',CHR(27),'PrSM2',CHR(27),'\');
end; {reset+scroll}


PROCEDURE delay;
begin
        writeln ('!s(t255)');
end;  {delay}


PROCEDURE gigi+wake+up;
        {will get terminal ready for graphics commands}

begin
        writeln ('!');
end;  {gigi+wake+up}



PROCEDURE set+text+macros;
        {will write gigi macros to be used in indicating
        size and color of characters in text}
{not reliable}
begin
        {text size 1 - standard}
        writeln ('!m,1;Mt(sl,w(v,i(r)))3;');     {red}
        writeln ('!m,0;Mt(sl,w(v,i(b)))5;');     {blue}



end; {set+text+macros}
```

```
PROCEDURE gigi_init;
        {will call routines to change default set ups
        and establish text macros}

begin
        set_up_gigi;
        set_text_macros;
end; {gigi_init}



PROCEDURE clear_screen;
        {will erase screen}

begin
        gigi_wake_up;
        writeln ('!s(e)');
end;    {clear_screen}




PROCEDURE draw_screen;
        {will erase screen, set background to dark,
        and set shading references to draw horizontal
        lines at 429 and 451 and a vertical line
        at 548}

begin
        clear_screen;
        writeln ('!w(r,s[,451],i(w))');
        writeln ('!p[0,452] v [+767,]');
        writeln ('!w(r,s[,429],i(w))');
        writeln ('!p[0,428] v [+767,]');
end; {draw_screen}


PROCEDURE message_posit;
        {will move cursor to position to start message}

begin
        gigi_wake_up;
        writeln ('!p[0,430]');
end;  {message_posit}


PROCEDURE clear_msg;

         {will erase message by resetting shading reference
         and color and refilling message area}

begin
        gigi_wake_up;
        writeln ('!w(e,s[,450],i(d))');
        message_posit;
```

```
        writeln ('!v[+767,]');
end; {clear*msg}


PROCEDURE clear*display;
        {will erase data entry display by resetting
        shading reference and color and refilling
        data display area}
begin
        gigi*wake*up;
        writeln ('!w(e,s[,427],i(d))');
        writeln ('!p[0,0]  v[+547,]');
end; {clear*display}


PROCEDURE clear*menu;
        {will erase menu area by resetting shading reference
        and color and refilling menu area}
begin
        gigi*wake*up;
        writeln ('!w(e,s[,427],i(d))');
        writeln ('!p[550,0]  v[+210,]');
end;  {clear*menu}


PROCEDURE clear*input;
        {will erase input area by resetting shading
        reference and color in  refilling input area}
begin
        gigi*wake*up;
        writeln ('!w(r,s[,453],i(d))');
        writeln ('!p[0,479]  v[+767,]');
end;  {clear*input}
```

```
{file text.pas - contains title and instruction screens;
        help screens may be added here later}


PROCEDURE continue*posit;
        {will move cursor to position for continue messages in instructions}
begin
        writeln ('!p[20,590]');
end;  {continue*posit}


PROCEDURE continue*text;
        {will write continue instructions at continue position}
begin
        continue*posit;
        writeln ('!t(s1,w(v,i(c)))''Type "y <cr>" to continue');
end;  {continue*text}


PROCEDURE title*text;
        {will write title screen to start execution of program}

begin
  clear*screen;

  writeln ('!p[186,100]');
  writeln ('!t(s2,w(v,i(a)))''COMPUTER AIDED DESIGN');

  writeln ('!p[375,150]');
  writeln ('!t(s1,w(v,i(a)))''of');

  writeln ('!p[160,200]');
  writeln ('!t(s2,w(v,i(a)))''MICROPROCESSOR-BASED CONTROL SYSTEMS');

  continue*posit;
  writeln ('!t(s1,w(v,i(c)))''type "y <cr>" for instructions, "n <cr>" to begin data entry');

end;  {title*text}


PROCEDURE instruct*text*1;
        {will write instructions to screen}

begin
  clear*screen;

  writeln ('!p[20,60]');
  writeln ('!t(s1,w(v,i(c)))''Data will be entered in five areas:');

  writeln ('!p[40,90]');
  writeln ('!t(s1,w(v,i(c)))''1.  Identification');

  writeln ('!p[40,110]');
  writeln ('!t(s1,w(v,i(c)))''2.  Design Criteria');

  writeln ('!p[40,130]');
```

```
  writeln (!t(sl,w(v,i(c)))''5.  Contingency/Task Pairs');

  writeln (!p[40,150]');
  writeln (!t(sl,w(v,i(c)))''4.  Functions');

  writeln (!p[40,170]');
  writeln (!t(sl,w(v,i(c)))''5.  Procedures');

  writeln (!p[20,200]');
  writeln (!t(sl,w(v,i(c)))''You will be able to select the area in which you wish to work.');

  continue+text;


end;  {instruct+text+1}


PROCEDURE instruct+text+2;

begin
  clear+screen;

  writeln (!p[20,60]');
  writeln (!t(sl,w(v,i(c)))''Menus will be used to prompt for data entry.');

  writeln (!p[20,90]');
  writeln (!t(sl,w(v,i(c)))''Type in either the data for the item requested');

  writeln (!p[20,120]');
  writeln (!t(sl,w(v,i(c)))''or the number of the choice selected, as appropriate.');

  writeln (!p[20,170]');
  writeln (!t(sl,w(v,i(c)))''Identifiers (subroutine and variable names) should be limited ');

  writeln (!p[20,200]');
  writeln (!t(sl,w(v,i(c)))''to ten characters and should begin with a letter.');

  writeln (!p[20,230]');
  writeln (!t(sl,w(v,i(c)))''Comment and identification data lines are 80 characters.');

  writeln (!p[20,280]');
  writeln (!t(sl,w(v,i(c)))''A <cr> alone will result in a zero value for numerical');

  writeln (!p[20,310]');
  writeln (!t(sl,w(v,i(c)))''data or a string of blanks for character data.');

  writeln (!p[20,340]');
  writeln (!t(sl,w(v,i(c)))''It can also be used to advance the marker when changing data.');

  continue+text;

end;  {instruct+text+2}



PROCEDURE instruct+text+3;

begin
  clear+screen;
```

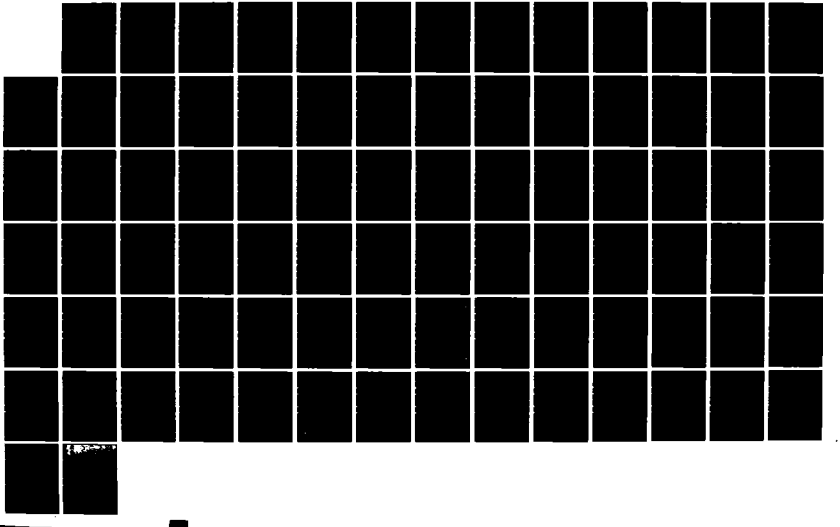
```
    writeln (':p[20,100]');
    writeln (':t(s),w(v,i(c)))''Following data entry you will have three options:');

    writeln (':p[60,130]');
    writeln (':t(s),w(v,i(c)))''1.  Change data already entered');

    writeln (':p[60,150]');
    writeln (':t(s),w(v,i(c)))''2.  Convert data entered to primitive list format');

    writeln (':p[60,170]');
    writeln (':t(s),w(v,i(c)))''3.  Write data entered to a file');

    writeln (':p[60,190]');
    writeln (':t(s),w(v,i(c)))''4.  Exit');

    continue:=posit;
    writeln (':t(s),w(v,i(c)))''Type "y <cr>" to begin');

  end;  {instruct:text:3}
```

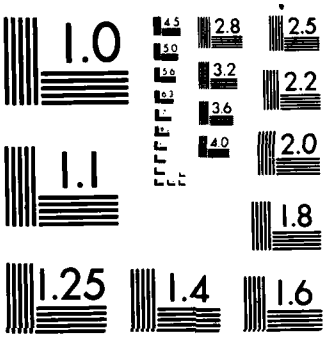
```
{file messages.pas - message subroutine}

   var
        menum : integer;


procedure message (menum : integer);
        {will move cursor to the message position
        and write the message indicated by the
        message number menum
        messages limited to 80 characters}


begin
  messageposit;

  case menum of
    1:
      writeln ('!t(sl,w(v,i(r)))''choice not in menu - please reenter');

    2:
      writeln ('!t(sl,w(v,i(r)))''already done');

    3:
      writeln ('!t(sl,w(v,i(r)))''option not yet implem  'ed');

    4:
      writeln ('!t(sl,w(v,i(r)))''enter "y <cr>" to confirm exit, "n <cr>" to continue design');

    5:
      writeln ('!t(sl,w(v,i(r)))''identifier contains unacceptable characters - please reenter');

    6:
      writeln ('!t(sl,w(v,i(r)))''contingency/task pair with name given already entered');

    7:
      writeln ('!t(sl,w(v,i(r)))''subroutine name already used');

    8:
      writeln ('!t(sl,w(v,i(r)))''subroutine already entered');

    9:
      writeln ('!t(sl,w(v,i(r)))''please enter either "y" or "n" ');

   10:
      writeln ('!t(sl,w(v,i(r)))''please enter "y" ');

   11:
      writeln ('!t(sl,w(v,i(r)))''more than six digits - please reenter');

   12:
      writeln ('!t(sl,w(v,i(r)))''non-numeric characters used - please reenter');

   13:
      writeln ('!t(sl,w(v,i(r)))''conversion error - please reenter');

   14:
      writeln ('!t(sl,w(v,i(r)))''enter "y <cr>" to make changes, "n <cr>" to exit section');
```

```
    15:
       writeln ('!t(sl,w(v,i(r)))''entry contains more than 10 characters - please reenter');

    16:
       writeln ('!t(sl,w(v,i(r)))''entry contains blanks - please reenter');

    17:
       writeln ('!t(sl,w(v,i(r)))''no name entered - please reenter');

    18:
       writeln ('!t(sl,w(v,i(r)))''variable data already entered');


  end; {case}
end; {message}

{end file messages.pas}
```

AD-A132 085 USER-FRIENDLY SYNTAX DIRECTED INPUT TO A COMPUTER AIDED DESIGN SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA B J SHERLOCK JUN 83 2/2

UNCLASSIFIED F/G 9/2 NL





```
{file menus.pas - menu routines used in design.pas}


{BASIC SETTINGS}


{menu lines limited to 22 characters}
{instruction lines limited to 24 characters}



var
        line : integer;




PROCEDURE menuline (line : integer);
        {will set cursor position to start of menu
        line passed to it as parameter}

begin

        case line of
                0:
                        {heading}
                        writeln ('!p[560,30]');
                1:
                        writeln ('!p[570,100]');
                2:
                        writeln ('!p[570,130]');
                3:
                        writeln ('!p[570,160]');
                4:
                        writeln ('!p[570,190]');
                5:
                        writeln ('!p[570,220]');
                6:
                        writeln ('!p[570,250]');
                7:
                        writeln ('!p[570,280]');

        end;  {case}

end;  {menuline}



PROCEDURE subheading;
        {will set marker position to subheading position}
```

```
begin
        writeln ('!p[570,501');
end;



PROCEDURE marker+line (line : integer);
        {will set marker position for the menu line
        passed to it}

begin

        case line of

                1:
                        writeln ('!p[550,100]');
                2:
                        writeln ('!p[550,130]');
                3:
                        writeln ('!p[550,160]');
                4:
                        writeln ('!p[550,190]');
                5:
                        writeln ('!p[550,220]');
                6:
                        writeln ('!p[550,250]');
                7:
                        writeln ('!p[550,280]');

        end;  {case}

end;  {marker+line}



PROCEDURE instruction+line (line : integer);
        {will position cursor for the menu instruction
        line passed to it}

begin

        case line of

                1:
                        writeln ('!p[550,310]');
                2:
                        writeln ('!p[550,330]');
                3:
                        writeln ('!p[550,350]');
                4:
                        writeln ('!p[550,370]');

        end;  {case}

end;  {instruction+line}

PROCEDURE draw+marker (line : integer);
        {will draw a marker '>' on the menu line passed
        to it, at the standard x coordinate for markers}
```

```
begin

        gigi+wake+up;
        marker+line (line);
        writeln ('!t(s1,w(v,i(v)))''>');
end; {draw+marker}


PROCEDURE clear+marker (line : integer);
        {will erase the marker at the menu line passed
        to it}

begin
        gigi+wake+up;
        marker+line (line);
        writeln ('!t(s1,w(v,i(d)))'' ');
end; {clear+marker}



{INSTRUCTION MENUS}


PROCEDURE menu+instructions+1;
        {will give instructions for entering data consisting
        of character strings}

begin

        instruction+line (1);
        writeln ('!t(s1,w(v,i(w)))''type in data item');

        instruction+line (2);
        writeln ('!t(s1,w(v,i(w)))''indicated by ">", ');

        instruction+line (3);
        writeln ('!t(s1,w(v,i(w)))''followed by <cr>');
end; {menu+instructions+1}


PROCEDURE menu+instructions+1A;
        {will give instructions for  exiting menu}

begin
        instruction+line (4);
        writeln ('!t(s1,w(v,i(w)))''<cr> to exit ');
end;  {menu+instructions+1A}



PROCEDURE menu+instructions+1B+unary;
        {will give instructions to avoid entering left operand for
        unary operator}
begin
```

```
        instruction+line (4);
        writeln (':t(sl,w(v,i(w)))''<cr> if unary expression');
end;  {menu+instructions+1B+unary}


PROCEDURE menu+instructions+2;
        {will give instructions for entering menu choices}

begin
        instruction+line (1);
        writeln (':t(sl,w(v,i(w)))''type number of menu item');

        instruction+line (2);
        writeln (':t(sl,w(v,i(w)))''followed by <cr>');

        instruction+line (3);
        writeln (':t(sl,w(v,i(w)))''to enter choice');
end;  {menu+instructions+2}


PROCEDURE menu+instructions+3;
        {will write instructions for monitors and volumes data entry}
begin
        instruction+line (1);
        writeln (':t(sl,w(v,i(w)))''type number for priority');

        instruction+line (2);
        writeln (':t(sl,w(v,i(w)))''in which monitor/volume');

        instruction+line (3);
        writeln (':t(sl,w(v,i(w)))''should be examined');
end;  {menu+instructions+3}


PROCEDURE change+instructions+1;
        {will tell how to move cursor and enter changed data}
begin
        instruction+line (1);
        writeln (':t(sl,w(v,i(w)))''type <cr> to move ">" ');

        instruction+line (2);
        writeln (':t(sl,w(v,i(w)))''down to item to change');
end;  {change+instructions+1}


PROCEDURE change+instructions+1A;
        {will tell how to enter changed data}
begin
        instruction+line (3);
        writeln (':t(sl,w(v,i(w)))''enter new value or name');

        instruction+line (4);
        writeln (':t(sl,w(v,i(w)))''followed by <cr>');
end;  {change+instructions+1A}
```

```
{GENERAL MENU COMPONENTS}


PROCEDURE data_areas_block;
        {list of 5 data areas and exit, to be called by entry and
        change main menus}
begin
        menu_line (1);
        writeln ('!t(sl,w(v,i(c)))''1. identification');

        menu_line (2);
        writeln ('!t(sl,w(v,i(c)))''2. design criteria');

        menu_line (3);
        writeln ('!t(sl,w(v,i(c)))''3. contingencies/tasks');

        menu_line (4);
        writeln ('!t(sl,w(v,i(c)))''4. procedures');

        menu_line (5);
        writeln ('!t(sl,w(v,i(c)))''5. functions');

        menu_line (6);
        writeln ('!t(sl,w(v,i(c)))''6. exit');
end;  {data_areas_block}


PROCEDURE timing_units_menu;
        {will display a standard list of timing unit
        options available}
begin
        menu_line (1);
        writeln ('!t(sl,w(v,i(c)))''1.  hours  (h)');

        menu_line (2);
        writeln ('!t(sl,w(v,i(c)))''2.  minutes  (m)');

        menu_line (3);
        writeln ('!t(sl,w(v,i(c)))''3.  seconds  (s)');

        menu_line (4);
        writeln ('!t(sl,w(v,i(c)))''4.  milliseconds  (ms)');

        menu_line (5);
        writeln ('!t(w(v,i(c)))''5.  microseconds  (us)');


        menu_line (6);
        writeln ('!t(sl,w(v,i(c)))''6.  nanoseconds  (ns)');

end;  {timing_units_menu}



{DATA ENTRY AND CHANGE MENUS}
```

```
        {MAIN}

PROCEDURE main_entry_menu;
        {will give choices for data entry areas}

begin
        clear_menu;

        menu_line (0);
        writeln ('!t(s1,w(v,i(m)))''DATA ENTRY AREAS');

        data_areas_block;

        menu_instructions_2;

end; {main_entry_menu}


PROCEDURE main_change_menu;
         {will list choices for data change areas}
begin
        clear_menu;

        menu_line (0);
        writeln ('!t(s1,w(v,i(m)))''DATA CHANGE AREAS');

        data_areas_block;

        menu_instructions_2;
end;  {main_change_menu}


        {IDENTIFICATION DATA MENU}


PROCEDURE id_menu;

        {will write id information menu on the screen}

begin
        clear_menu;

        {heading}
        menu_line (0);
        writeln ('!t(s1,w(v,i(m)))''DESIGN IDENTIFICATION');

        menu_line (1);
        writeln ('!t(s1,w(v,i(c)))''designer name');

        menu_line (2);
        writeln ('!t(s1,w(v,i(c)))''design title');

        menu_line (3);
        writeln ('!t(s1,w(v,i(c)))''design date');
```

```
        menu*line (4);
        writeln (':t(s),w(v,i(c)))''comments');

        menu*instructions*1;
        menu*instructions*1A;

end;  {id*menu}

        {CRITERIA DATA MENUS}

PROCEDURE criteria*menu*heading;

        {will write standard heading for all criteria menus}
begin
        menu*line (0);
        writeln (':t(s),w(v,i(m)))''CRITERIA');
end;  {criteria*menu*heading}


PROCEDURE criteria*menu*crit;
        {will write menu of criteria options available}
begin
        clear*menu;

        criteria*menu*heading;

        menu*line (1);
        writeln (':t(s),w(v,i(c)))''1.  first');

        menu*line (2);
        writeln (':t(s),w(v,i(c)))''2.  cost');

        menu*line (3);
        writeln (':t(s),w(v,i(c)))''3.  power');

        menu*instructions*2;
end;  {criteria*menu*crit}



PROCEDURE criteria*menu*mon;
        {will write monitor menu}
begin
        clear*menu;

        criteria*menu*heading;

        subheading;
        writeln (':t(s),w(v,i(w)))''monitors');

        menu*line (1);
        writeln (':t(s),w(v,i(c)))''monitor 1');

        menu*line (2);
        writeln (':t(s),w(v,i(c)))''monitor 2');

        menu*line (3);
        writeln (':t(s),w(v,i(c)))''monitor 3');
```

```
        menu+line (4);
        writeln (':t(s),w(v,i(c)))''monitor 4');

        menu+line (5);
        writeln (':t(s),w(v,i(c)))''monitor 5');

        menu+instructions+3;
end;  {criteria+menu+mon}


PROCEDURE criteria+menu+vol1;
        {will write volume menu}
begin
        clear+menu;

        criteria+menu+heading;

        subheading;
        writeln (':t(s),w(v,i(w)))''volumes 1-5');

        menu+line (1);
        writeln (':t(s),w(v,i(c)))''volume 1');

        menu+line (2);
        writeln (':t(s),w(v,i(c)))''volume 2');

        menu+line (3);
        writeln (':t(s),w(v,i(c)))''volume 3');

        menu+line (4);
        writeln (':t(s),w(v,i(c)))''volume 4');

        menu+line (5);
        writeln (':t(s),w(v,i(c)))''volume 5');

        menu+instructions+3;
end;  {criteria+menu+vol1}


PROCEDURE criteria+menu+vol2;
        {will write volume menu}
begin
        clear+menu;

        criteria+menu+heading;

        subheading;
        writeln (':t(s),w(v,i(w)))''volumes 6-10');

        menu+line (1);
        writeln (':t(s),w(v,i(c)))''volume 6');

        menu+line (2);
        writeln (':t(s),w(v,i(c)))''volume 7');

        menu+line (3);
        writeln (':t(s),w(v,i(c)))''volume 8');

        menu+line (4);
```

```
        writeln (':t(s),w(v,i(c)))''volume 9');

        menu+line (5);
        writeln (':t(s),w(v,i(c)))''volume 10');

        menu+instructions+3;
end;  {criteria+menu+vol2}


PROCEDURE criteria+menu+comments;
        {will write menu for comments for criteria section}
begin
        clear+menu;

        criteria+menu+heading;

        menu+line (1);
        writeln (':t(s),w(v,i(c)))''comments');

        menu+instructions+1;
        menu+instructions+1A;
end;  {criteria+menu+comments}



        {CONTINGENCY/TASK PAIR DATA MENUS}


PROCEDURE ct+menu+heading;
        {will write standard heading for all ct menus}
begin
        menu+line (0);
        writeln (':t(s),w(v,i(m)))''CONTINGENCY/TASK PAIRS');
end;  {ct+menu+heading}


PROCEDURE ct+menu+comments;
        {will display menu to obtain comments regarding
        c/t pair}
begin
        clear+menu;

        ct+menu+heading;

        menu+line (1);
        writeln (':t(s),w(v,i(c)))''comments');

        menu+instructions+1;
        menu+instructions+1A;

end;  {ct+menu+comments}


PROCEDURE ct+menu+type;
        {will display menu to determine type of c/t pair}
begin
```

```
        clear_menu;

        ct_menu_heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''        type');

        menu_line (1);
        writeln ('!t(s1,w(v,i(c)))''1.  initialization');

        menu_line (2);
        writeln ('!t(s1,w(v,i(c)))''2.  run time');

        menu_instructions_2;

end;  {ct_menu_type}



PROCEDURE ct_menu_names;
        {will write menu to obtain function and
        procedure names}

begin
        clear_menu;

        ct_menu_heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''  subroutine names');

        menu_line (1);
        writeln ('!t(s1,w(v,i(c)))''function (contingency)');

        menu_line (2);
        writeln ('!t(s1,w(v,i(c)))''procedure (task)');


        menu_instructions_1;

end;  {ct_menu_names}



PROCEDURE ct_menu_time_units;
        {will display menu of timing unit options}
begin
        clear_menu;

        ct_menu_heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''    timing units');

        timing_units_menu;

        menu_instructions_2;
```

```
end;  {ct+menu+time+units}



PROCEDURE ct+menu+cont+def;
        {will display menu for contingency definition}
begin
        clear+menu;

        ct+menu+heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''contingency definition');

        menu+line (1);
        writeln ('!t(s1,w(v,i(c)))''1.  when');

        menu+line (2);
        writeln ('!t(s1,w(v,i(c)))''2.  at');

        menu+line (3);
        writeln ('!t(s1,w(v,i(c)))''3.  every');

        menu+line (4);
        writeln ('!t(s1,w(v,i(c)))''4.  do');

        menu+instructions+2;

end;  {ct+menu+cont+def}



PROCEDURE ct+values+block;
        {menu of values to be entered, used with both entry and
        change menus}
begin
        subheading;
        writeln ('!t(s1,w(v,i(w)))''       values');

        menu+line (1);
        writeln ('!t(s1,w(v,i(c)))''rho');

        menu+line (2);
        writeln ('!t(s1,w(v,i(c)))''beta1');

        menu+line (3);
        writeln ('!t(s1,w(v,i(c)))''beta2');

        menu+line (4);
        writeln ('!t(s1,w(v,i(c)))''global order');

        menu+line (5);
        writeln ('!t(s1,w(v,i(c)))''local priority');

        menu+line (6);
        writeln ('!t(s1,w(v,i(c)))''gamma');

        menu+line (7);
```

```
        writeln ('!t(s1,w(v,i(c)))''gamma2');

end;  {ct*values*block}



PROCEDURE ct*menu*values;
        {will display menu for various parameters}
begin
        clear*menu;

        ct*menu*heading;
        ct*values*block;

        menu*instructions*1;

end;  {ct*menu*values}



PROCEDURE ct*change*menu*values;
        {will display menu with change instructions for various parameters}
begin
        clear*menu;

        ct*menu*heading;
        ct*values*block;

        change*instructions*1;
        change*instructions*1A;
end;  {ct*change*menu*values}



PROCEDURE ct*menu*background;
        {will display menu to obtain boolean value
        for background}
begin
        clear*menu;

        ct*menu*heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''     background');

        menu*line (1);
        writeln ('!t(s1,w(v,i(c)))''1.  true');

        menu*line (2);
        writeln ('!t(s1,w(v,i(c)))''2.  false');

        menu*instructions*2;

end;  {ct*menu*background}
```

```
PROCEDURE ct*menu*condition;
        {will display menu of possible conditions
        on contingency/task action}
begin
        clear*menu;

        ct*menu*heading;

        subheading;
        writeln ('!t(sl,w(v,i(w)))''     condition');

        menu*line (1);
        writeln ('!t(sl,w(v,i(c)))''1.  unless');

        menu*line (2);
        writeln ('!t(sl,w(v,i(c)))''2.  if');

        menu*line (3);
        writeln ('!t(sl,w(v,i(c)))''3.  none');

        menu*instructions*2;

end;  {ct*menu*condition}


PROCEDURE ct*change*menu*areas;
        {will write menu giving areas in which to make changes}
begin
        clear*menu;

        ct*menu*heading;

        subheading;
        writeln ('!t(sl,w(v,i(w)))''  change areas');

        menu*line (1);
        writeln ('!t(sl,w(v,i(c)))''1. type/subroutines');

        menu*line (2);
        writeln ('!t(sl,w(v,i(c)))''2. data/comments');

        menu*line (3);
        writeln ('!t(sl,w(v,i(c)))''3. exit');

        menu*instructions*2;
end;  {ct*change*menu*areas}


PROCEDURE ct*change*menu*data;
        {will write menu giving data items which can be changed}
begin
        clear*menu;

        ct*menu*heading;

        subheading;
        writeln ('!t(sl,w(v,i(w)))''  data items');
```

```
        menu+line (1);
        writeln ('!t(sl,w(v,i(c)))''1. timing units');

        menu+line (2);
        writeln ('!t(sl,w(v,i(c)))''2. contingency defn.');

        menu+line (3);
        writeln ('!t(sl,w(v,i(c)))''3. data values');

        menu+line (4);
        writeln ('!t(sl,w(v,i(c)))''4. background');

        menu+line (5);
        writeln ('!t(sl,w(v,i(c)))''5. condition');

        menu+line (6);
        writeln ('!t(sl,w(v,i(c)))''6. comments');

        menu+line (7);
        writeln ('!t(sl,w(v,i(c)))''7. exit');

        menu+instructions+2;
end;  {ct+change+menu+data}



        {SUBROUTINE MENUS}


PROCEDURE subroutine+heading;
        {will write heading for subroutine menus}
begin
        menu+line (0);
        writeln ('!t(sl,w(v,i(m)))''SUBROUTINE DATA');
end;  {subroutine+heading}



PROCEDURE subroutine+menu;
        {will write menu for subroutine name and comments}
begin
        clear+menu;

        subroutine+heading;

        menu+line (1);
        writeln ('!t(sl,w(v,i(c)))''subroutine name');

        menu+line (2);
        writeln ('!t(sl,w(v,i(c)))''comments');

        menu+instructions+1;
        menu+instructions+4;
end;  {subroutine+menu}
```

```
PROCEDURE statements+menu+type;
        {will give menu of types of statements which can be entered}
begin
        clear+menu;

        subroutine+heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''  statement type');

        menu+line (1);
        writeln ('!t(s1,w(v,i(c)))''1. assignment');

        menu+line (2);
        writeln ('!t(s1,w(v,i(c)))''2. conditional');

        menu+line (3);
        writeln ('!t(s1,w(v,i(c)))''3. input/output');

        menu+line (4);
        writeln ('!t(s1,w(v,i(c)))''4. exit');

        menu+instructions+2;

end;  {statements+menu+type}


PROCEDURE statements+menu+io;
        {will write menu listing options for input/output statements}
begin
        clear+menu;

        subroutine+heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''input/output statement');

        menu+line (1);
        writeln ('!t(s1,w(v,i(c)))''1. input (sense)');

        menu+line (2);
        writeln ('!t(s1,w(v,i(c)))''2. output (issue)');

        menu+line (3);
        writeln ('!t(s1,w(v,i(c)))''3. exit');

        menu+instructions+2;
end;  {statements+menu+io}



PROCEDURE statements+menu+expression;
        {will write menu of options for expression - either expression
        or terminal (variable or constant)}
begin
        clear+menu;
```

```
        subroutine_heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''statement expression');

        menu_line (1);
        writeln ('!t(s1,w(v,i(c)))''1. term (vbl or const)');

        menu_line (2);
        writeln ('!t(s1,w(v,i(c)))''2. expression');

        menu_instructions_2;
end;  {statements_menu_expression}


PROCEDURE statements_menu_ops;
        {will write menu of operators available}
begin
        clear_menu;

        subroutine_heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''statement operators');

        menu_line (1);
        writeln ('!t(s1,w(v,i(c)))''1. add (+)');

        menu_line (2);
        writeln ('!t(s1,w(v,i(c)))''2. subtract (-)');

        menu_line (3);
        writeln ('!t(s1,w(v,i(c)))''3. multiply (*)');

        menu_line (4);
        writeln ('!t(s1,w(v,i(c)))''4. divide (/)');

        menu_line (5);
        writeln ('!t(s1,w(v,i(c)))''5. none (no operator)');

        menu_instructions_2;
end;  {statements_menu_ops}



PROCEDURE statements_menu_variable_name;
        {will write menu to prompt for entry of variable name}
begin
        clear_menu;

        subroutine_heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''  variable name');

        menu_line (1);
        writeln ('!t(s1,w(v,i(c)))''variable name');

        menu_instructions_1;
```

```
end;  {staements_menu_variable_name}


%include 'menus2.pas/nolist'



{DATA MANIPULATION MENUS}

PROCEDURE phase_two_menu;
        {will give menu of options for data entered in
        phase one}

begin
        clear_menu;

        menu_line (0);
        writeln ('!t(s),w(v,i(m)))''DESIGN DATA OPTIONS');

        menu_line (1);
        writeln ('!t(s),w(v,i(c)))''1. change data');

        menu_line (2);
        writeln ('!t(s),w(v,i(c)))''2. generate primitives');

        menu_line (3);
        writeln ('!t(s),w(v,i(c)))''3. write to file');

        menu_line (4);
        writeln ('!t(s),w(v,i(c)))''4. exit');

        menu_instructions_2;

end; {phase_two_menu}


{end menus.pas}
```

```
{file menus2.pas - second file of menus - contains menus for
variable data}

        {VARIABLE DATA MENUS}



PROCEDURE variable_heading;
        {will write heading for all variable menus}
begin
        menu_line (0);
        writeln ('!t(s1,w(v,i(m)))''VARIABLE DATA');
end;  {variable_heading}


PROCEDURE variable_menu_types;
        {will write menu of six major types in which variables can
        be classified}
begin
        clear_menu;

        variable_heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''   type');

        menu_line (1);
        writeln ('!t(s1,w(v,I(c)))''1. analog');

        menu_line (2);
        writeln ('!t(s1,w(v,i(c)))''2. boolean');

        menu_line (3);
        writeln ('!t(s1,w(v,i(c)))''3. external fixed');

        menu_line (4);
        writeln ('!t(s1,w(v,i(c)))''4. external float');

        menu_line (5);
        writeln ('!t(s1,w(v,i(c)))''5. internal fixed');

        menu_line (6);
        writeln ('!t(s1,w(v,i(c)))''6. internal float');

        menu_instructions_2;
end;  {variable_menu_types}



PROCEDURE variable_menu_scope;
        {will write menu to determine if variable is global or local}
begin
        clear_menu;

        variable_heading;
```

```
        subheading;
        writeln ('!t(s1,w(v,i(w)))'' scope');

        menu+line (1);
        writeln ('!t(s1,w(v,i(c)))''1. global');

        menu+line (2);
        writeln ('!t(s1,w(v,i(c)))''2. local');

        menu+instructions+2;
end;  {variable+menu+scope}


PROCEDURE variable+menu+technology;
        {will write menu of types of technology available}
begin
        clear+menu;

        variable+heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))'' technology');

        menu+line (1);
        writeln ('!t(s1,w(v,i(c)))''1. ttl');

        menu+line (2);
        writeln ('!t(s1,w(v,i(c)))''2. ecl');

        menu+line (3);
        writeln ('!t(s1,w(v,i(c)))''3. i2l');

        menu+instructions+2;
end;  {variable+menu+technology}


PROCEDURE variable+menu+addr;
        {will write menu to obtain address of variable}
begin
        clear+menu;

        variable+heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))'' location');

        menu+line (1);
        writeln ('!t(s1,w(v,i(c)))''memory address');

        menu+instructions+1;
end;  {variable+menu+addr}


PROCEDURE variable+menu+precision+1;
        {will write menu for precision of 8,16,32,64}
```

```
begin
        clear+menu;

        variable+heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''  precision');

        menu+line (1);
        writeln ('!t(s1,w(v,i(c)))''1. 8');

        menu+line (2);
        writeln ('!t(s1,w(v,i(c)))''2. 16');

        menu+line (3);
        writeln ('!t(s1,w(v,i(c)))''3. 32');

        menu+line (4);
        writeln ('!t(s1,w(v,i(c)))''4. 64');

        menu+instructions+2;
end;  {variable+menu+precision+1}


PROCEDURE variable+menu+precision+2;
        {will write menu for precision of 16,32,64}
begin
        clear+menu;

        variable+heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''  precision');

        menu+line (1);
        writeln ('!t(s1,w(v,i(c)))''1. 16');

        menu+line (2);
        writeln ('!t(s1,w(v,i(c)))''2. 32');

        menu+line (3);
        writeln ('!t(s1,w(v,i(c)))''3. 64');

        menu+instructions+2;
end;  {variable+menu+precision+2}


PROCEDURE variable+menu+port;
        {will write menu to obtain port for variable}
begin
        clear+menu;

        variable+heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''   port');

        menu+line (1);
        writeln ('!t(s1,w(v,i(c)))''port number');
```

```
        menu^instructions^1;
end;  {variable^menu^oort}


PROCEDURE variable^menu^sample^units;
        {will write menu to get timing units for sampling}
begin
        clear^menu;

        variable^heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))'' sample rate');

        timing^units^menu;

        menu^instructions^2;
end;  {variable^menu^sample^units}



PROCEDURE variable^menu^sample^cycle;
        {will write menu to get timing value for sampling}
begin
        clear^menu;

        variable^heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))'' sample rate');

        menu^line (1);
        writeln ('!t(s1,w(v,i(c)))'' amount of time');

        menu^instructions^1;
end;  {variable^menu^sample^cycle}


PROCEDURE variable^menu^internal^type;
        {will write menu to determine whether internal variable
        is a variable or constant}
begin
        clear^menu;

        variable^heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))'' internal type');

        menu^line (1);
        writeln ('!t(s1,w(v,i(c)))''1. variable');

        menu^line (2);
        writeln ('!t(s1,w(v,i(c)))''2. constant');

        menu^instructions^2;
end;  {variable^menu^internal^type}
```

```
PROCEDURE variable*menu*init*val;
        {will write menu to obtain initialization value for variable}
begin
        clear*menu;

        variable*heading;

        subheading;
        writeln (':t(s1,w(v,i(w)))'' initialization');

        menu*line (1);
        writeln (':t(s1,w(v,i(c)))''initial value');

        menu*instructions*1;
end;  {variable*menu*init*val}


PROCEDURE variable*menu*bool*init;
        {will write menu to obtain initialization value for
        boolean variables}
begin
        clear*menu;

        variable*heading;

        subheading;
        writeln (':t(s1,w(v,i(w)))'' initialization');

        menu*line (1);
        writeln (':t(s1,w(v,i(c)))''1. true (1)');

        menu*line (2);
        writeln (':t(s1,w(v,i(c)))''2. false (0)');

        menu*instructions*2;
end;  {variable*menu*bool*init}



PROCEDURE variable*menu*max*min;
        {will write menu to obtain maximum and minimum values for variable}
begin
        clear*menu;

        variable*heading;

        subheading;
        writeln (':t(s1,w(v,i(w)))'' max/min values');

        menu*line (1);
        writeln (':t(s1,w(v,i(c)))''minimum value');

        menu*line (2);
        writeln (':t(s1,w(v,i(c)))''maximum value');
```

```
        menu+instructions+1;
end;  {variable+menu+max+min}



PROCEDURE variable+menu+float;
        {will write menu to obtain exponent and mantissa size for
        floating point variables}
begin
        clear+menu;

        variable+heading;

        subheading;
        writeln ('!t(s1,w(v,i(w)))''floating point data');

        menu+line (1);
        writeln ('!t(s1,w(v,i(c)))''mantissa size (bits)');

        menu+line (2);
        writeln ('!t(s1,w(v,i(c)))''exponent size (bits)');

        menu+instructions+1;
end;  {variable+menu+float}


{end menus2.pas}
```

```
{file display.pas - conatins subroutines which will write out data in
        records to display area of screen}

var
        dline : integer;


PROCEDURE display+line (dline : integer);
        {will move cursor to display line positions}
begin
   case dline of
        0:      writeln ('!p[40,20]');

        1:      writeln ('!p[20,40]');

        2:      writeln ('!p[20,60]');

        3:      writeln ('!p[20,80]');

        4:      writeln ('!p[20,100]');

        5:      writeln ('!p[20,120]');

        6:      writeln ('!p[20,140]');

        7:      writeln ('!p[20,160]');

        8:      writeln ('!p[20,180]');

        9:      writeln ('!p[20,200]');

       10:      writeln ('!p[20,220]');

       11:      writeln ('!p[20,240]');

       12:      writeln ('!p[20,260]');

       13:      writeln ('!p[20,280]');

       14:      writeln ('!p[20,300]');

       15:      writeln ('!p[20,320]');

       16:      writeln ('!p[20,340]');

       17:      writeln ('!p[20,360]');

       18:      writeln ('!p[20,380]');

       19:      writeln ('!p[20,400]');
   end;  {case}
end;  {display+lines}


PROCEDURE reset+display+line (var line : integer);
        {will reset line counter to 1 if it exceeds 19}
begin
        if line > 19 then
          begin
            line := 1;
```

```
            clear_display;
          end;
end;



PROCEDURE display_instructions;
        {will write exit/change instruction at the bottom line in
        the display area}
begin
        display_line (19);
        writeln ('!t(sl,w(v,i(q)))''enter "y <cr>" to call change routine, "n <cr>" to exit');
end;  {display_instructions}



PROCEDURE display_comments (var i : integer; first_comt : comt_ptr);
        {will write comments to display area, starting first line with
        'Comments:' and splitting lines which exceed the display area
        in length}
    const
        forty_blanks = '                                        ';
    var
        len : integer;
        temp : comt_ptr;
        line_one, line_two : packed array [1..40] of char;
        j : integer;
begin
  if first_comt = nil then
    begin
        display_line (i);
        writeln ('!t(sl,w(v,i(q)))''Comments:  none');
        i := i + 1;
    end
  else
    begin
        temp := first_comt;
        len := length (temp^.line);
        if len > 40 then
          begin
                line_two := forty_blanks;
                for j := 1 to 40 do
                  line_one [j] := temp^.line [j];
                for j := 41 to len do
                  line_two [j - 40] := temp^.line [j];
                display_line (i);
                writeln ('!t(sl,w(v,i(q)))''Comments:  ',line_one,'-');
                i := i + 1;
                display_line (i);
                writeln ('!t(sl,w(v,i(q)))''           ',line_two);
          end
        else
          begin
                display_line (i);
                writeln ('!t(sl,w(v,i(q)))''Comments:  ',temp^.line);
          end;
        i := i + 1;
        temp := temp^.next_comt;
        while temp <> nil do
```

```
            begin
                  len := length (tempf.line);
                  if len > 40 then
                     begin
                       line+two := forty+blanks;
                       for j := 1 to 40 do
                          line+one [j] := tempf.line [j];
                       for j := 41 to len do
                          line+two [j - 40] := tempf.line [j];
                       display+line (i);
                       writeln ('!t(s],w(v,i(g)))''            ',line+one,'-');
                       i := i + 1;
                       display+line (i);
                       writeln ('!t(s],w(v,i(g)))''            ',line+two);
                     end
                  else
                     begin
                       display+line (i);
                       writeln ('!t(s],w(v,i(g)))''            ',tempf.line);
                     end;
                  i := i + 1;
                  temp := tempf.next+comt;
            end;
      end;
end;  {display+comments}



PROCEDURE display+id+data (id+data : id+ptr);
        {will write contents of id record to display area}
   var
        i : integer;
begin
        clear+display;

        display+line (1);
        writeln ('!t(s],w(v,i(g)))''Designer:  ',id+dataf.designer);

        display+line (2);
        writeln ('!t(s],w(v,i(g)))''Project:   ',id+dataf.project);

        display+line (5);
        writeln ('!t(s],w(v,i(g)))''Date:      ',id+dataf.date);

        i := 4;
        display+comments (i,id+dataf.comt+link);
end;  {display+id+data}



PROCEDURE display+criteria+data (crit+data : criteria+ptr);
        {will write data in criteria, monitors, and volumes records to
        the display area}
   var
        i : integer;
begin
        clear+display;

        display+line (1);
```

```
        writeln ('!t(s1,w(v,i(q)))''Criteria:  ',crit^data^.crit);

        display^line (2);
        with crit^data^.monitors^link^ do
          writeln ('!t(s1,w(v,i(q)))''Monitors: ',m1:4,m2:4,m3:4,m4:4,m5:4);

        display^line (3);
        with crit^data^.volumes^link^ do
          writeln ('!t(s1,w(v,i(q)))''Volumes:  ',v1:4,v2:4,v3:4,v4:4,v5:4,v6:4,v7:4,v8:4,v9:4,v10:4);

        i := 4;
        display^comments (i,crit^data^.comt^link);
end;  {display^criteria^data}



PROCEDURE display^ct^pairs^data (ct^data : ct^pair^ptr);
        {will write data in contingency/task pair record to the
        display area}
    var
        i : integer;
begin
        clear^display;

        display^line (1);
        case ct^data^.kind of
          init:
            writeln('!t(s1,w(v,i(q)))''Kind:  Initialization');
          run:
            writeln ('!t(s1,w(v,i(q)))''Kind:  Run Time');
        end;  {case}

        display^line (2);
        writeln ('!t(s1,w(v,i(q)))''Function:  ',ct^data^.funct.name);

        display^line (3);
        writeln ('!t(s1,w(v,i(q)))''Procedure: ',ct^data^.proct.name);

        display^line (4);
        writeln ('!t(s1,w(v,i(q)))''Timing units:  ',ct^data^.time^units);

        display^line (5);
        writeln ('!t(s1,w(v,i(q)))''Contingency definition:  ',ct^data^.cont^def);

        display^line (6);
        writeln ('!t(s1,w(v,i(q)))''rho = ',ct^data^.rho);

        display^line (7);
        writeln ('!t(s1,w(v,i(q)))''beta1 = ',ct^data^.beta1);

        display^line (8);
        writeln ('!t(s1,w(v,i(q)))''beta2 = ',ct^data^.beta2);

        display^line (9);
        writeln ('!t(s1,w(v,i(q)))''global order = ',ct^data^.glob^ord);

        display^line (10);
        writeln ('!t(s1,w(v,i(q)))''local priority = ',ct^data^.loc^pri);

        display^line (11);
```

```
        writeln ('it(s),w(v,i(q)))''gamma1 = ',ct^.data^.gamma1);

        display^line (12);
        writeln ('it(s),w(v,i(q)))''gamma2 = ',ct^.data^.gamma2);

        display^line (13);
        writeln ('it(s),w(v,i(q)))''background = ',ct^.data^.background);

        display^line (14);
        writeln ('it(s),w(v,i(q)))''condition = ',ct^.data^.ct^.cond);

        i := 15;
        display^comments (i,ct^.data^.comt^.link);
end;  {display^ct^pair^data}



        {VARIABLE DISPLAY ROUTINES}


PROCEDURE display^curr^variable (curr^variable : expr^item^ptr);
        {will display name of current variable in list of variables used
        in a subroutine}
begin
        clear^display;

        display^line (1);
        writeln ('it(s),w(v,i(q)))''variable = ', curr^variable^.name);
end;  {display^curr^variable}



PROCEDURE display^ext^float^vbl (var^data : vbl^ptr);
        {will display data entered for a variable of type
        external floating point}
begin
        display^line (2);
        writeln ('it(s),w(v,i(q)))''type = external floating point');

        display^line (3);
        writeln ('it(s),w(v,i(q)))''initial value =', var^data^.value);

        display^line (4);
        writeln ('it(s),w(v,i(q)))''memory address =', var^data^.mem^addr);

        display^line (5);
        writeln ('it(s),w(v,i(q)))''precision =',var^data^.precision^efl);

        display^line (6);
        writeln ('it(s),w(v,i(q)))''minimum value =',var^data^.min^val^efl);

        display^line (7);
        writeln ('it(s),w(v,i(q)))''maximum value =',var^data^.max^val^efl);

        display^line (8);
        writeln ('it(s),w(v,i(q)))''mantissa size =',var^data^.mantis^efl);

        display^line (9);
        writeln ('it(s),w(v,i(q)))''exponent size =',var^data^.expon^efl);
```

```
        display_line (10);
        writeln ('!t(s1,w(v,i(q)))''port number =',var_data^.port_num_ef1);
end;  {display_ext_float_vbl}



PROCEDURE display_variable (var_data : vbl_ptr);
        {will display name of variable for which data has been entered and
        then call the appropriate subroutine, determined by the variable type,
        to display the rest of the data}

begin
        clear_display;

        display_line (1);
        writeln ('!t(s1,w(v,i(q)))''variable =',var_data^.name);

        case var_data^.kind1 of
           analog:
                message (3);     {not yet implemented}
           booln:
                message (3);
           ext_fixed:
                message (3);
           ext_float:
                display_ext_float_vbl (var_data);
           int_fixed:
                message (3);
           int_float:
                message (3);
        end;  {case}
end;  {display_variable}


PROCEDURE display_begin (var line : integer);
        {will write 'begin' on the screen; to be used in conjunction
        with display of statements}
begin
        display_line (line);
        writeln ('!t(s1,w(v,i(q)))''begin');
        line := line +1;
        reset_display_line (line);
end;



PROCEDURE display_end (var line : integer);
        {will write 'end;' on the screen;  to be used in conjunction
        with display of statements}
begin
        display_line (line);
        writeln ('!t(s1,w(v,i(q)))''end;');
        line := line + 1;
        reset_display_line (line);
end;
```

```
PROCEDURE display*io*stmt (var line : integer; statement*ptr : stmt*ptr);
        {will write io statement data from record to screen
        display area, increment line counter and, if it exceeds 19, reset it}
   var
        kind : (sense, issue);
begin
        if statement*ptr <> nil then
          begin
             display*line (line);
             case statement*ptr*.direction*.operator of
                data*in:        {input}
                        kind := sense;
                data*out:       {output}
                        kind := issue;
             end;  {case}
             writeln (':t(s1,w(v,i(q)))'' ',kind,'(',statement*ptr*.direction*.right*ptr*.name,')};');
             line := line + 1;
             reset*display*line (line);
          end;
end;  {display*io*stmt}


PROCEDURE disolay*statements (var line : integer; statmts*ptr : stmt*ptr);
        {will traverse list of statements and display them until
        it reaches the end of the list}
   var
        statement*ptr : stmt*ptr;
begin
        statement*ptr := statmts*ptr;
        if statement*ptr = nil then
           writeln (':t(s1,w(v,i(q)))''no statements entered')
        else
         begin
           display*begin (line);
           repeat  {until statement*ptr = nil}
                case statement*ptr*.stmt*type of
                  io:
                        display*io*stmt (line,statement*ptr);
                   {add others later}
                end;  {case}
                statement*ptr := statement*ptr*.next*stmt;
           until statement*ptr = nil;
           display*end (line);
         end;
end;  {display*statements}



PROCEDURE disolay*subroutine*name (var line : integer; sr*data : subrout*ptr);
        {will write subroutine type and name on given line}
begin
        disolay*line (line);
        if sr*data*.kind = func then
           writeln (':t(s1,w(v,i(q)))''function ', sr*data*.name,';')
        else  {procedure}
```

```
            writeln (';t(sl,w(v,i(q)))''Procedure ', sr^data^.name,';');
          line := line + 1;
end;  {display^subroutine^name}



PROCEDURE display^subroutine (sr^data : subrout^ptr);
        {will write subroutine data to display area of screen}
    var
        line : integer;
begin
        clear^display;
        line := 1;
        display^subroutine^name (line, sr^data);
        display^comments (line, sr^data^.comt^link);
        display^statements (line, sr^data^.stmts^link);
end;  {display^subroutine}
```

```
{this is check.pas}

{file check.pas - primitive check routines for entering or
changing data}


{BASIC ROUTINES}

PROCEDURE check*item (var item : name*size; temp : line*size; var done*item : boolean);
        {will take identifier entered by user and check length,
        that first character is a letter, and that there are no blanks}

   const
        alphabet = ['a'..'z'];

   var
        item*error : boolean;
        len : integer;
        i : integer;

begin
           item*error := false;
           len := length (temp);
           if len = 0 then
             begin
                message (17);  {no name entered}
                item*error := true;
             end
           else if len > 10 then
             begin
                message (15);  {name too long}
                item*error := true;
             end
           else if not (temp[1] in alphabet) then
             begin
                message (5);
                item*error := true;
             end
           else
             begin
                for i := 1 to len do
                   if temp[i] = ' ' then
                         item*error := true;
                if item*error then
                         message (16);  {contains blanks}
             end;
           if not item*error then
             begin
                for i := 1 to len do
                   item[i] := temp[i];
                if len < 10 then
                   for i := len + 1 to 10 do
                      item[i] := ' ';
                done*item := true;
             end
           else  {was error}
                item := 'item*error';
```

```
end;  {check_item}


PROCEDURE check_digits (var conv_val : integer; number : digits; var done_digits : boolean);
        {will check numbers entered as character strings for non-numeric characters,
        including blanks, and for size greater than 6 digits.  If errors
        exist, will write appropriate message and await reentry.
        If there are no errors, will call system library routine to convert
        the character string to its numerical value.}

   const
        number_set = ['0'..'9'];
   type
        six_digits = packed array [1..7] of char;
   var
         digit_error : boolean;
        len, result : integer;
        number_six : six_digits;
        i : integer;
        value : integer;


   [ASYNCHRONOUS,EXTERNAL (OTS$CVI_TI_L)] FUNCTION $cvt_ti_l (%stdescr inp_str :[volatile]
        six_digits; %ref value : [volatile] integer; value_size : integer := %immed 4;
        flags : integer := %immed 1) : integer;
        external;

begin
          digit_error := false;
          len := length (number);
          if len = 0 then
            begin
                conv_val := 0;
                done_digits := true;
            end
          else if len > 6 then
            begin
                message (11);  {more than 6 digits}
                digit_error := true;

            end
          else
            begin
                for i := 1 to len do
                  if not (number[i] in number_set) then
                     digit_error := true;
                if digit_error then
                     message (12);  {non-numeric characters}

             end;
           if ((not digit_error) and (not done_digits)) then
             begin
                number_six [1] := '+';
                for i := 1 to len  do
                    number_six [i + 1] := number [i];
                if len < 6 then
                    for i := len + 2 to 7 do
                        number_six [i] := ' ';
                result := $cvt_ti_l (%stdescr number_six, %ref value);
                if odd (result) then
```

```
                begin
                  con^val := value;
                  done^digits := true;
                end
              else
                  message (13);  {conversion error}
            end;
          if not done^digits then
              con^val := -999;
end;  {check^digits}



PROCEDURE subroutine^name^check (var temp^ptr : subrout^ptr;
                                 name : name^size);
        {will traverse a linked list of subroutines until it
        finds a name match or reaches the last record in
        the list}

begin
        if temp^ptr <> nil then
          while (( temp^ptr^.next <> nil ) and
                  not(temp^ptr^.name = name)) do
                temp^ptr := temp^ptr^.next;
end;  {subroutine^name^check}



PROCEDURE ct^name^check (var temp^ptr : ct^pair^ptr;
                         func^name, proc^name : name^size);
        {will traverse a linked list of c/t pair records
        until it finds a function or procedure name match or
        reaches the last record in the list}

begin
        if temp^ptr <> nil then
          while ((temp^ptr^.next^pair <> nil) and
                 not((func^name = temp^ptr^.funct.name) and
                  not(proc^name = temp^ptr^.proct.name)) do
            temp^ptr := temp^ptr^.next^pair;
end;  {ct^name^check}


PROCEDURE check^variable^list(var var^ptr : vbl^ptr; name : name^size);
        {will traverse a linked list of variable data records until it finds
        a name match or reaches the last record in the list}
begin
```

```
        if varptr <> nil then
                while ((varptrf.nextvbl <> nil ) and
                       (varptrf.name <> name)) do
                    varptr := varptrf.nextvbl;
end;  {checkvariablelist}



{end check.pas}
```

```
{file enter.pas - pascal subroutines for entering data}


{BASIC ROUTINES}

PROCEDURE enter_item (var item : name_size);
        {will read identifier entered by user and pass it to check_item
        to check for various errors}

   var
        temp : line_size;
        done_item : boolean;

begin
        done_item := false;
        repeat  {until done_item true}
           clear_input;
           readln (temp);
           clear_msg;
           check_item (item, temp, done_item);
        until done_item;
        clear_input;
end;  {enter_item}


PROCEDURE enter_digits (var con_val : integer);
        {will read value entered by user and call check_digits to check
        entry for errors}

   var
        done_digits : boolean;
        number : digits;


begin
        done_digits := false;
        repeat  {until} done_digits true}
          clear_input;
          readln (number);
          clear_msg;
          check_digits (con_val, number, done_digits);
        until done_digits;
        clear_input;
end;  {enter_digits}



PROCEDURE header_record (var design_data : design_ptr);

        {will get design record and initialize all
        pointer fields in it to nil; will also initialize
        ct_hook and env_hook to nil}
```

```
  var
       ct^head : ct^types^ptr;

begin
       new (design^data);
       new (ct^head);
       with design^data^ do
         begin
              id^link := nil;
              ct^link := ct^head;
              crit^link := nil;
              env^link := nil;
              func^link := nil;
              proc^link := nil;

         end;  {with}

       with ct^head^ do
         begin
              init^list := nil;
              run^list := nil;
         end;  {with}

end;  {header^record}
```



```
PROCEDURE enter^comments (var first^comt : comt^ptr);
       {will create a linked list of lines of comments,
       with first^comt pointing to the first record
       in the list}

   var
       c, cnext : comt^ptr;
       done^comts : boolean;
       comments : line^size;

begin
  readln (comments);
  if length (comments) = 0 then
    first^comt := nil

  else
    begin
       new (c);
       first^comt := c;
       c^.line := comments;
       done^comts := false;
       clear^input;


       repeat {until done^comts}
         readln (comments);
         if length (comments) = 0 then
           begin
              done^comts := true;
```

```
                cf.next^comt := nil;
            end
          else
            begin
                new(cnext);
                cf.next^comt := cnext;
                c := cf.next^comt;
                cf.line := comments;
            end;

          clear^input;

        until done^comts;

    end;  {else}
end;  {enter^comts}
```



```
{SECTION ENTRY ROUTINES}

        {IDENTIFICATION}


PROCEDURE enter^id (var id^link : id^ptr);

        {will display id menu and instructions, and then
        obtain identification information, placing
        it in the fields of the id record and linking
        the id record to the design record.  Will then display data entered and
        allow designer the opportunity to call the section change routine
        to modify the data.}

   var
        name, day, title, comments : line^size;
        id^data : id^ptr;
        choice : char;
        done^id : boolean;


begin
        id^menu;

        draw^marker(1);
        readln (name);
        clear^marker(1);

        draw^marker (2);
        clear^input;
        readln (title);
        clear^marker (2);

        draw^marker (3);
```

```
        clear_input;
        readln (day);
        clear_marker (3);

        new(id_data);

        with id_data^ do
           begin
                designer := name;
                project := title;
                date := day;
           end; {with}

        draw_marker (4);
        clear_input;
        enter_comments (id_data^.com_link);

        id_link := id_data;

        display_id_data (id_link);
        display_instructions;
        done_id := false;
        repeat  {until done_id}
          readln (choice);
          clear_msg;
          case choice of
            'n':  {exit}
                done_id := true;
            'y':  {change}
                message (3);
            otherwise
                message (9);  {enter y or n}
          end;  {case}
        until done_id;

end;  {enter_id}


        {CONTINGENCY/TASK}


PROCEDURE check_other_ct_list (var ct_done : boolean;
        ct_type : ct_kind; type_ptr : ct_types_ptr;
        func_name, proc_name : name_size);
        {given  a kind of c/t pair of either init or run,
        will check the other kind list to see if the name
        has already been used;  if it has, will return
        ct_done of true}

   var
        other_list_ptr : ct_pair_ptr;

begin
        case ct_type of
          init:
                other_list_ptr := type_ptr^.run_list;

          run:
                other_list_ptr := type_ptr^.init_list;
        end;  {case}
```

```
        ct^.name^.check (other^.list^.ptr,func^.name,proc^.name);
        if other^.list^.ptr <> nil then
           if ((other^.list^.ptr^.funct.name = func^.name) or
               (other^.list^.ptr^.proct.name = proc^.name)) then
           ct^.done := true;
end;  {check^.other^.ct^.list}


PROCEDURE add^.to^.ct^.list (var ct^.temp : ct^.pair^.ptr;
        var ct^.data : ct^.pair^.ptr; var ct^.done : boolean;
        func^.name, proc^.name : name^.size);

        {will traverse a list of c/t records to match either
        function or procedure names.  if either match, will
        return true for  ct^.done.  otherwise will add a
        new c/t pair record}

begin
             ct^.name^.check (ct^.temp, func^.name, proc^.name);
             if ((ct^.temp^.funct.name = func^.name) or
                 (ct^.temp^.proct.name = proc^.name)) then
               begin
                 ct^.done := true;
                 ct^.data := nil;
               end
             else
               begin
                 while ct^.temp^.next^.pair <> nil do
                   ct^.temp := ct^.temp^.next^.pair;
                 new (ct^.data);
                 ct^.temp^.next^.pair := ct^.data;
               end;
end;  {add^.to^.ct^.list}


PROCEDURE add^.ct^.record (var ct^.data : ct^.pair^.ptr;
        var ct^.done : boolean; ct^.type : ct^.kind;
        type^.ptr : ct^.types^.ptr; func^.name, proc^.name : name^.size);
        {if no records are in the appropriate type list, will
        place the first record on the list.  otherwise will call add^.to^.ct^.list
        check appropriate list and, if c/t pair not already
        entered, will add record ; will then enter c/t type}

    var
        ct^.temp : ct^.pair^.ptr;

begin

  case ct^.type of
    init:
      begin
        if type^.ptr^.init^.list = nil then
          begin
            new (ct^.data);
            type^.ptr^.init^.list := ct^.data;
          end
        else
          begin
            ct^.temp := type^.ptr^.init^.list;
```

```
            add_to_ct_list(ct_temp,ct_data,ct_done,func_name,proc_name);
          end;
      end;
    run:
      begin
        if type_ptr^.run_list = nil then
          begin
            new(ct_data);
            type_ptr^.run_list := ct_data;
          end
        else
          begin
            ct_temp := type_ptr^.run_list;
            add_to_ct_list(ct_temp,ct_data,ct_done,func_name,proc_name);
          end;
      end;
    end;  {case}
    if not ct_done then
      ct_data^.kind := ct_type;

end;  {add_ct_record}



PROCEDURE ct_subroutine_link (var s_ptr : subrout_ptr;
        s_name : name_size; s_type : subrout_kind;
        ct_data : ct_pair_ptr; design_data : design_ptr);
        {will check procedure or function list against the
        procedure or function name used with the ct pair.
        if a match is found, ct and subroutine records
        will be linked.  if not, a new subroutine record
        will be obtained, linked, and initialized}

   var
        s_data : subrout_ptr;

begin
  if s_ptr = nil then
    begin
        new (s_data);
        case s_type of
          func :
                design_data^.func_link := s_data;
          proc :
                design_data^.proc_link := s_data;
        end;  {case}

        with s_data^ do
          begin
                kind := s_type;
                name := s_name;
                ct_pair := ct_data;
                comt_link := nil;
                first_vpl := nil;
                loc_vtl_link := nil;
                stmts_link := nil;
                next := nil;
          end;  {with}
```

```
    end  {if}

  else
    begin
        subroutine_name_check (s_ptr,s_name);
        if s_ptr^.name = s_name then
          begin
            s_data := s_ptr;
            s_data^.ct_pair := ct_data;
          end

        else
         begin
          while s_ptr^.next <> nil do
            s_ptr := s_ptr^.next;
          new (s_data);
          s_ptr^.next := s_data;

          with s_data^ do
            begin
                 kind := s_type;
                 name := s_name;
                 ct_pair := ct_data;
                 com_link := nil;
                 first_vbl := nil;
                 loc_vbl_link := nil;
                 stmts_link := nil;
                 next := nil;
            end;  {with}
         end;  {else}
    end;  {else}
    case s_type of
        proc:
          ct_data^.proc := s_data;
        func:
          ct_data^.func := s_data;
    end;  {case}

end;  {ct_subroutine_link}



PROCEDURE enter_ct_type (var ct_type : ct_kind);
        {will determine type of c/t pair, initialization
        or run time, to be entered}

    var
        choice : char;
        done_type : boolean;

begin
        ct_menu_type;
        done_type := false;
        repeat  {until done_type}
          readln (choice);
          clear_msg;
          case choice of
            'i':
                begin
                  ct_type := init;
```

```
                done^type := true;
              end;
          '2':
              begin
                ct^type := run;
                done^type := true;
              end;
           otherwise
             begin
              message (1);
              clear^input;
             end;
         end;  {case}

      until done^type;
      clear^input;

end;    {enter^ct^type}



PROCEDURE enter^ct^names (ct^type : ct^kind;
      design^data : design^ptr; var ct^done : boolean;
      var ct^data : ct^pair^ptr);
      {will obtain function and procedure names, call check^other^ct^list to
      to see that the pair has not already been entered
      in the other list (init or run), and, if not, call
      add^ct^record to check names and add a record to the
      given c/t list, and call ct^subroutine^link to
      link it to the function and procedure named}

    var
      ct^temp : ct^pair^ptr;
      type^ptr : ct^types^ptr;
      func^name, proc^name : name^size;
      func^ptr, proc^ptr : subrout^ptr;

begin
  {initialize pointers to be used}
  type^ptr := design^data^.ct^link;
  func^ptr := design^data^.func^link;
  proc^ptr := design^data^.proc^link;

  ct^done := false;

  ct^menu^names;

  draw^marker (1);
  enter^item (func^name);
  clear^marker (  ;
  clear^input;

  draw^marker (2);
  enter^item (proc^name);
  clear^input;

  check^other^ct^list (ct^done,ct^type,type^ptr,func^name,proc^name);

  if not ct^done then
```

```
    add_ct_record(ct_data,ct_done,ct_type,type_ptr,func_name,proc_name);

  if not ct_done then
    begin
        ct_subroutine_link (func_ptr,func_name,func,ct_data,design_data);
        ct_subroutine_link (proc_ptr,proc_name,proc,ct_data,design_data);
    end;

end;  {enter_ct_names}


procedure enter_ct_time_units (ct_data : ct_pair_ptr);

   var
        choice : char;
        done_units : boolean;
begin
        ct_menu_time_units;
        repeat  {until done_units true}
          readln (choice);
          clear_msg;
          case choice of
            '1':
               begin
                ct_data^.time_units := h;
                done_units := true;
               end;

            '2':
               begin
                ct_data^.time_units := m;
                done_units := true;
               end;

            '3':
               begin
                ct_data^.time_units := s;
                done_units := true;

               end;

            '4':
               begin
                ct_data^.time_units := ms;
                done_units := true;
               end;

            '5':
               begin
                ct_data^.time_units := us;
                done_units := true;
               end;

            '6':
               begin
                ct_data^.time_units := ns;
                done_units := true;
               end;

            otherwise
```

```
                begin
                 message (1);
                 done_units := false;
                 clear_input;
                end;
            end;  {case}

        until done_units;
        clear_input;

end;  {enter_ct_time_units}



PROCEDURE enter_ct_contin_def (ct_data : ct_pair_ptr);

    var
        choice : char;
        done_def : boolean;

begin

        ct_menu_cont_def;
        repeat  {until done_def = true}
          readln (choice);
          clear_msg;
          case choice of
            '1':
                begin
                 ct_data^.cont_def := when;
                 done_def := true;
                end;

            '2':
                begin
                 message (3);
                 done_def := false;
                end;

            '3':
                begin
                 message (3);
                 done_def := false;
                end;

            '4':
                begin
                 message (3);
                 done_def := false;
                end;

            otherwise
                begin
                 message (1);
                 done_def := false;
                 clear_input;
                end;
          end;  {case}

        until done_def;
```

```
        clear_input;

end;  {enter_ct_contingdef}



PROCEDURE enter_ct_background (ct_data : ct_pair_ptr);

    var
        choice : char;
        done_bg : boolean;

begin
        ct_menu_background;
        repeat  {until done_bg true}
          readln (choice);
          clear_msg;
          case choice of
            '1':
               begin
                ct_data^.background := true;
                done_bg := true;
               end;

            '2':
               begin
                ct_data^.background := false;
                done_bg := true;
               end;

            otherwise
               begin
                message (1);
                done_bg := false;
                clear_input;
               end;

          end;  {case}

        until done_bg;
        clear_input;

end;  {enter_ct_background}



PROCEDURE enter_ct_condition (ct_data : ct_pair_ptr);

   var
        choice : char;
        done_cond : boolean;

begin
        ct_menu_condition;
        repeat  {until done_condition = true}
          readln (choice);
          clear_msg;
          case choice of
            '1':
```

```
                    begin
                     message (3);
                     done+cond := false;
                    end;

                '2':
                    begin
                     message (3);
                     done+cond := false;
                    end;

                '3':
                    begin
                     ct+data1.ct+cond := none;
                     done+cond := true;
                    end;

                otherwise
                    begin
                     message (1);
                     done+cond := false;
                     clear+input;
                    end;

              end;  (case)

          until done+cond;
          clear+input;

end;  (enter+ct+condition)


PROCEDURE enter+ct+values (ct+data : ct+pair+ptr);

   var
        t+rho,t+bet1,t+bet2,t+alob,t+locl : integer;
        t+gam1, t+gam2 : integer;

begin
        ct+menu+values;

        draw+marker (1);
        enter+digits (t+rho);
        clear+marker (1);

        draw+marker (2);
        enter+digits (t+bet1);
        clear+marker (2);

        draw+marker (3);
        enter+digits (t+bet2);
        clear+marker (3);

        draw+marker (4);
        enter+digits (t+alob);
        clear+marker (4);

        draw+marker (5);
        enter+digits (t+locl);
        clear+marker (5);
```

```
        draw*marker (6);
        enter*digits (t*gam1);
        clear*marker (6);

        draw*marker (7);
        enter*digits (t*gam2);
        clear*marker (7);

        with ct*data^ do
          begin
                rho := t*rho;
                beta1 := t*bet1;
                beta2 := t*bet2;
                glob*ord := t*glob;
                loc*pri := t*loc;
                gamma1 := t*gam1;
                gamma2 := t*gam2;
          end;

end;  {enter*ct*values}



PROCEDURE enter*ct*comments (ct*data : ct*pair*ptr);

begin
        ct*menu*comments;
        draw*marker (1);
        enter*comments (ct*data^.comt*link);
end;  {enter*ct*comments}



PROCEDURE enter*ct*pairs (design*data : design*ptr);
        {will call subroutines to enter data in
        fields of c/t pair record.  will then display dat entered
        and allow designer to call ct change subroutine to modify data}

   var
        ct*done : boolean;
        ct*type : ct*kind;
        ct*data : ct*pair*ptr;
        done*ct : boolean;
        choice : char;

begin
        enter*ct*type (ct*type);
        enter*ct*names (ct*type,design*data,ct*done,ct*data);

        if ct*done then
                message (6)
        else
          begin
                enter*ct*time*units(ct*data);
                enter*ct*contingency(ct*data);
                enter*ct*background (ct*data);
                enter*ct*condition (ct*data);
                enter*ct*values (ct*data);
```

```
                enter_ct_comments (ct_data);

                display_ct_pairs_data (ct_data);
                display_instructions;
                done_ct := false;
                repeat  {until done_ct}
                  readln (choice);
                  clear_msg;
                  case choice of
                    'n':  {exit}
                        done_ct := true;
                    'y':  {change}
                        begin
                          change_ct_pairs (ct_data);
                          done_ct := true;
                        end;
                    otherwise
                        message (9);  {enter y or n}
                  end;  {case}
                until done_ct;
            end;
end;  {enter_ct_pairs}


%include 'enter2.pas/nolist'


{MAIN ENTRY ROUTINE}

PROCEDURE enter_new_main (var design_data : design_ptr);
        {will get header record, display the main
        entry menu with instructions, and continue
        in a loop of calling entry subroutines until
        user desires to exit}


   var
        done_enter : boolean;
        choice : char;

begin
        draw_screen;
        header_record (design_data);
        done_enter := false;

        repeat  {until done_enter}

           clear_input;

           main_entry_menu;

           readln (choice);
           clear_msg;

           case choice of
                '1':
                    {id}
                    if design_data^.id_link = nil then
                       begin
                         clear_input;
```

```
                    enter_id (design_data^.id_link);
                  end
              else
                  message(2);  {already done}

          '2':
              {criteria}
              if design_data^.crite_link = nil then
                 begin
                  clear_input;
                  enter_criteria (design_data^.crite_link);
                 end
              else
                  message (2);  {already done}

          '3':
              begin
               clear_input;
               enter_ct_pairs (design_data);
              end;


          '4':
              {procedure}
              begin
                  clear_inout;
                  enter_subroutine (design_data, proc);
              end;

          '5':
              {function}
              begin
                  clear_input;
                  enter_subroutine (design_data, func);
              end;

          '6':
              {exit}
              done_enter := true;


          otherwise
              message(1);  {error msg}

        end;  {case}

        clear_display;
      until done_enter;

end;  {enter_new_main}

{end enter.pas}
```

```
{file enter2.pas - second file of entry routines - contains subroutine,
criteria, and variable related routines}


        {CRITERIA DATA ENTRY ROUTINES}

PROCEDURE enter_criteria_crit (crit_data : criteria_ptr);
        {will call menu and read criteria option selected
        by the user}
   var
        choice : char;
        done_crit : boolean;
begin
        criteria_menu_crit;
        done_crit := false;
        repeat  {until done_crit true}
          readln (choice);
          clear_msg;
          case choice of
            '1':
               begin
                crit_data^.crit := first;
                done_crit := true;
               end;
            '2':
               begin
                crit_data^.crit := cost;
                done_crit := true;
               end;
            '3':
               begin
                crit_data^.crit := power;
                done_crit := true;
               end;
            otherwise
               begin
                message (1);  {not in menu}
               end;
          end;  {case}
        until done_crit;
end;  {enter_criteria_crit}



PROCEDURE enter_criteria_comments (crit_data : criteria_ptr);
        {will  write menu and call enter_comments}
begin
        criteria_menu_comments;
        draw_marker (1);
        enter_comments (crit_data^.comt_link);
end;  {enter_criteria_comments}



PROCEDURE enter_criteria_monitors (var mon_link : monitors_ptr);
        {will write menu, obtain monitor data from user, get a new monitor record,
        enter the data, and link the record to the criteria record}
   var
```

```
        mon1,mon2,mon3,mon4,mon5 : integer;
        mon^data : monitors^ptr;
begin
        criteria^menu^mon;

        draw^marker (1);
        enter^digits (mon1);
        clear^marker (1);

        draw^marker (2);
        enter^digits (mon2);
        clear^marker (2);

        draw^marker (3);
        enter^digits (mon3);
        clear^marker (3);

        draw^marker (4);
        enter^digits (mon4);
        clear^marker (4);

        draw^marker (5);
        enter^digits (mon5);
        clear^marker (5);

        new (mon^data);
        with mon^data^ do
          begin
                m1 := mon1;
                m2 := mon2;
                m3 := mon3;
                m4 := mon4;
                m5 := mon5;
          end;
        mon^link := mon^data;
end;  {enter^criteria^monitors}


PROCEDURE enter^criteria^volumes (var vol^link : volumes^ptr);
        {will write menu, obtain volume data from user, get a new volume record,
        enter the data, and link the record to the criteria record}
   var
        vol1,vol2,vol3,vol4,vol5 : integer;
        vol6,vol7,vol8,vol9,vol10 : integer;
        vol^data : volumes^ptr;
begin
        criteria^menu^vol1;

        draw^marker (1);
        enter^digits (vol1);
        clear^marker (1);

        draw^marker (2);
        enter^digits (vol2);
        clear^marker (2);

        draw^marker (3);
        enter^digits (vol3);
        clear^marker (3);

        draw^marker (4);
```

```
        enter_digits (vol4);
        clear_marker (4);

        draw_marker (5);
        enter_digits (vol5);
        clear_marker (5);

        criteria_menu_vol2;

        draw_marker (1);
        enter_digits (vol6);
        clear_marker (1);

        draw_marker (2);
        enter_digits (vol7);
        clear_marker (2);

        draw_marker (3);
        enter_digits (vol8);
        clear_marker (3);

        draw_marker (4);
        enter_digits (vol9);
        clear_marker (4);

        draw_marker (5);
        enter_digits (vol10);
        clear_marker (5);


        new (vol_data);
        with vol_data^ do
          begin
                v1 := vol1;
                v2 := vol2;
                v3 := vol3;
                v4 := vol4;
                v5 := vol5;
                v6 := vol6;
                v7 := vol7;
                v8 := vol8;
                v9 := vol9;
                v10 := vol10;
          end;
        vol_link := vol_data;
end;  {enter_criteria_volumes}


PROCEDURE enter_criteria (var crit_link : criteria_ptr);
        {will get criteria record, call subroutines to get data, and then
        link criteria record to design record.  Will then display data
        entered and allow user opportunity to call change routine}
   var
        crit_data : criteria_ptr;
        done_crit : boolean;
        choice : char;
begin
        new (crit_data);
        enter_criteria_crit (crit_data);
        enter_criteria_comments (crit_data);
```

```
        enter_criteria_monitors (crit_data^.monitors_link);
        enter_criteria_volumes (crit_data^.volumes_link);
        crit_link := crit_data;

        display_criteria_data (crit_link);
        display_instructions;
        done_crit := false;
        repeat  {until done_crit}
          readln (choice);
          clear_msg;
          case choice of
            'n':  {exit}
                done_crit := true;
            'y':  {change}
                message (3);
            otherwise
                message (9);  {enter y or n}
          end;  {case}
        until done_crit;
end;  {enter_criteria}



{VARIABLE DATA ENTRY}

PROCEDURE enter_variable_scope (var scope : scope_types);
        {will display menu and obtain from the user whether the
        variable for which data will be entered is global or local
        in scope}
   var
        done_scope : boolean;
        choice : char;
begin
        variable_menu_scope;
        done_scope := false;
        repeat  {until done scope}
          clear_input;
          readln (choice);
          clear_msg;
          case choice of
            '1':
                begin
                  scope := global;
                  done_scope := true;
                end;
            '2':
                begin
                  scope := local;
                  done_scope := true;
                end;
            otherwise
                message (1);
          end;  {case}
        until done_scope;
end;  {enter_variable_scope}
```

```
PROCEDURE enter_variable_technology (var field : tech);
        {will obtain technology for variable from user}
   var
        done_tech : boolean;
        choice : char;
begin
        done_tech := false;
        variable_menu_technology;

        repeat  {until done_tech}
            clear_input;
            readln (choice);
            clear_msg;
            case choice of
              '1':
                  begin
                        field := ttl;
                        done_tech := true;
                  end;
                '2':
                  begin
                        field := ecl;
                        done_tech := true;
                  end;
                '3':
                  begin
                        field := i2l;
                        done_tech := true;
                  end;
              otherwise
                  message (1);
            end;  {case}
        until done_tech;
end;  {enter_variable_technology}


PROCEDURE enter_variable_addr (var field : integer);
        {will call menu and obtain value for address}
begin
        variable_menu_addr;
        draw_marker (1);
        enter_digits (field);
end;  {enter_variable_addr}


PROCEDURE enter_variable_precision (var field : precis);
        {will call menu and obtain value for precision of vairable}
   var
        choice : char;
        done_prec : boolean;
begin
        variable_menu_precision;
        done_prec := false;
        repeat  {until done_prec}
          clear_input;
          readln (choice);
          clear_msg;
```

```
          case choice of
            '1':
                begin
                  field := 8;
                  done_prec := true;
                end;
            '2':
                begin
                  field := 16;
                  done_prec := true;
                end;
            '3':
                begin
                  field := 32;
                  done_prec := true;
                end;
            '4':
                begin
                  field := 64;
                  done_prec := true;
                end;
            otherwise
                message (1);
          end;  {case}
        until done_prec;
end;  {enter_variable_precision_1}



PROCEDURE enter_variable_precision_2 (var field : precis2);
        {will call menu and obtain value for precision of vairable}
   var
        choice : char;
        done_prec : boolean;
begin
        variable_menu_precision_2;
        done_prec := false;
        repeat  {until done_prec}
          clear_input;
          readln (choice);
          clear_msg;
          case choice of
            '1':
                begin
                  field := 16;
                  done_prec := true;
                end;
            '2':
                begin
                  field := 32;
                  done_prec := true;
                end;
            '3':
                begin
                  field := 64;
                  done_prec := true;
                end;
            otherwise
                message (1);
```

```
          end;  {case}
        until done_prec;
end;  {enter_variable_precision_2}


PROCEDURE enter_variable_port (var field : integer);
        {will call menu and obtain value for port}
begin
        variable_menu_port;
        draw_marker (1);
        enter_digits (field);
end;  {enter_variable_port}


PROCEDURE enter_variable_sample_units (var field : units);
        {will call menu and obtain timing unit type}
    var
        choice : char;
        done_units : boolean;
begin
        variable_menu_sample_units;
        done_units := false;
        repeat  {until done_units}
           clear_input;
           readln (choice);
           clear_msg;
           case choice of
             '1':
                begin
                  field := h;
                  done_units := true;
                end;
             '2':
                begin
                  field := m;
                  done_units := true;
                end;
             '3':
                begin
                  field := s;
                  done_units := true;
                end;
             '4':
                begin
                  field := ms;
                  done_units := true;
                end;
             '5':
                begin
                  field := us;
                  done_units := true;
                end;
             '6':
                begin
                  field := ns;
                  done_units := true;
                end;
             otherwise
                message (1);
           end;  {case}
```

```
        until done_units;
        clear_input;
end;  {enter_variable_sample_units}



PROCEDURE enter_variable_sample_cycle (var field : integer);
        {will call menu and obtain amount of units of time for
        sampling cycle}
begin
        variable_menu_sample_cycle;
        draw_marker (1);
        enter_digits (field);
end;  {enter_variable_sample_cycle}


PROCEDURE enter_variable_internal_type (var field : internal_type);
        {will write menu and get internal variable type (variable or
        constant)}
   var
        done_type : boolean;
        choice : char;
begin
        done_type := false;
        variable_menu_internal_type;
        repeat  {until done_type}
          clear_input;
          readln (choice);
          clear_msg;
          case choice of
            '1':
                begin
                  field := vbl;
                  done_type := true;
                end;
              '2':
                begin
                  field := cons;
                  done_type := true;
                end;
               otherwise
                  message (1);
          end;  {case}
        until done_type;
end;  {enter_variable_internal_type}



PROCEDURE enter_variable_init_val (var field : integer);
        {will call menu and obtain initial value}
begin
        variable_menu_init_val;
        draw_marker (1);
        enter_digits (field);
end;  {enter_variable_init_val}
```

```
PROCEDURE enter_variable_bool_init (var field : boolean);
        {will call menu and obtain initial value for boolean variables}
  var
        done_bool : boolean;
        choice : char;
begin
        variable_menu_bool_init;
        done_bool := false;
        repeat  {until done_bool}
          clear_input;
          readln (choice);
          clear_msg;
          case choice of
            '1':
                begin
                  field := true;
                  done_bool := true;
                end;
            '2':
                begin
                  field := false;
                  done_bool := true;
                end;
            otherwise
                message (1);
          end;  {case}
        until done_bool;
end;  {enter_variable_bool_init}


PROCEDURE enter_variable_max_min (var minimum : integer; var maximum : integer);
        {will call menu and obtain maximum and minimum values
        for variable}
begin
        variable_menu_max_min;
        draw_marker (1);
        enter_digits (minimum);
        clear_marker (1);

        draw_marker (2);
        enter_digits (maximum);
end;  {enter_variable_max_min}


PROCEDURE enter_variable_float (var mantissa : integer; var exponent : integer);
        {will call menu and obtain exponent and mantissa values
        for floating point variable}
begin
        variable_menu_float;
        draw_marker (1);
        enter_digits (mantissa);
        clear_marker (1);

        draw_marker (2);
        enter_digits (exponent);
end;  {enter_variable_float}
```

```
PROCEDURE enter_ext_float_vbl (variable_data : vbl_ptr);
        {will obtain data for external floating point variable and
        enter in appropriate record fields}
begin
        variable_data^.kind1 := ext_float;
        enter_variable_init_val (variable_data^.value);
        enter_variable_addr (variable_data^.mem_addr);
        enter_variable_precision_2 (variable_data^.precision_efl);
        enter_variable_max_min (variable_data^.min_val_efl,variable_data^.max_val_efl);
        enter_variable_technology (variable_data^.technology_efl);
        enter_variable_float (variable_data^.mantis_efl,variable_data^.expon_efl);
        enter_variable_port (variable_data^.port_num_efl);
end;  {enter_ext_float_vbl}



PROCEDURE enter_variable_type (variable_data : vbl_ptr);
        {will display menu to obtain variable type and, based on type
        entered, will call appropriate subroutine to enter rest of
        variable data}
   var
        choice : char;
        done_type : boolean;
begin
        variable_menu_types;
        done_type := false;
        repeat  {until done_type}
           clear_input;
           readln (choice);
           clear_msg;
           case choice of
             '1':       {analog}
                message (3);  {not yet implemented}
             '2':       {boolean}
                message (3);
             '3':       {external fixed}
                message (3);
             '4':       {external float}
                begin
                  enter_ext_float_vbl (variable_data);
                  done_type := true;
                end;
             '5':       {internal fixed}
                message (3);
             '6':       {internal float}
                message (3);
             otherwise
                message (1);
           end;  {case}
        until done_type;
end;  {enter_variable_type}


PROCEDURE enter_variable_data (variable_data : vbl_ptr; name : name_size);
        {will enter name into record provided, call enter_variable_type
        to obtain the rest of the data, and then display data
```

```
        entered and allow the user to make changes
        (change option not yet implemented))
    var
        done_vbl : boolean;
        choice : char;
begin
    variable_data^.name := name;
    enter_variable_type (variable_data);
    display_variable (variable_data);
    display_instructions;
    done_vbl := false;
    repeat  (until done_vbl)
        readln (choice);
        clear_msg;
        case choice of
          'n':  (exit)
                done_vbl := true;
          'y':  (change)
                message (3);  (not yet implemented)
          otherwise
                message (9);
        end;  (case)
     until done_vbl;

end;  (enter_variable_data)
```



```
PROCEDURE enter_variables (design_data : design_ptr; sr_data : subrout_ptr);
        (will traverse list of variables used in a subroutine, checking
        to see if data has already been entered, and if not, getting a
        new variable record, linking it and entering the name, and
        calling enter_variable_data to obtain and enter the rest of the
        information)
    var
        scope : scope_types;
        curr_variable : expr_item_ptr;
        var_ptr : vbl_ptr;
        variable_data : vbl_ptr;
begin
        curr_variable := sr_data^.first_vbl;
        while curr_variable <> nil do
          begin
                display_curr_variable (curr_variable);
                enter_variable_scope (scope);
                if scope = global then
                  if design_data^.env_link = nil then
                    begin
                        new (variable_data);
                        design_data^.env_link := variable_data;
                        enter_variable_data (variable_data,curr_variable^.name);
                        var_ptr := nil;
                    end
                  else
                    var_ptr := design_data^.env_link;
                if scope = local then
                  if sr_data^.loc_vbl_link = nil then
                    begin
                        new (variable_data);
```

```
                    sr^dataf.loc^vble^link := variable^data;
                    enter^variable^data (variable^data, curr^variablef.name);
                    var^ptr := nil;
                  end
                else
                  var^ptr := sr^dataf.loc^vble^link;
              if var^ptr <> nil then
                begin
                  check^variable^list (var^ptr,curr^variablef.name);
                  if var^ptrf.name <> curr^variablef.name then
                    begin
                      new (variable^data);
                      var^ptrf.next^vbl := variable^data;
                      enter^variable^data (variable^data, curr^variablef.name);
                    end
                  else
                    begin
                      message (18);   {already entered}
                      delay;
                      clear^msg;
                    end;
                end;  {if}
              curr^variable := curr^variablef.next^vbl;
          end;  {while}
end;  {enter^variables}
```



```
        {SUBROUTINE DATA ENTRY}

PROCEDURE check^other^sr^list (var name^used : boolean; sr^kind : subrout^kind;
     design^data : design^ptr; sr^name : name^size);
        {will check list of subroutine kind not given to see if
        name entered has been previously used}
   var
        other^list^ptr : subrout^ptr;
begin
        case sr^kind of
          proc:
                other^list^ptr := design^dataf.func^link;
          func:
                other^list^ptr := design^dataf.proc^link;
        end;  {case}

        subroutine^name^check (other^list^ptr, sr^name);

        if other^list^ptr <> nil then
          if other^list^ptrf.name = sr^name then
            begin
                name^used := true;
                message (7);
            end;
end;  {check^other^sr^list}


PROCEDURE add^to^sr^list (var sr^temp : subrout^ptr; var name^used : boolean;
```

```
    var sr*data : subrout*ptr; sr*kind : subrout*kind; sr*name : name*size);
        {will check list given for a name match.  if one is found will check statement
        link to see if entire subroutine has been entered; if so will set
        name*used to true and write an appropriate message, if not will set
        sr*data to that record.  otherwise will move to end of
        list, get new record, and initialize name, kind, ct*pair, and next fields.}

begin
        subroutine*name*check (sr*temp, sr*name);
        if sr*temp*.name = sr*name then
          if sr*temp*.stmts*link <> nil then
            begin
                message (A);
                name*used := true;
                sr*data := nil;
            end
          else
                sr*data := sr*temp
        else
          begin
            while sr*temp*.next <> nil do
                sr*temp := sr*temp*.next;
            new (sr*data);
            sr*temp*.next := sr*data;
            with sr*data* do
              begin
                kind := sr*kind;
                name := sr*name;
                ct*pair := nil;
                stmts*link := nil;
                next := nil;
              end;  {with}
          end;
end;  {add*to*sr*list}



PROCEDURE add*sr*record (var name*used : boolean; var sr*data : subrout*ptr;
              sr*kind : subrout*kind; design*data : design*ptr;
              sr*name : name*size);

        {will check given list and, if list has not yet been created, will
        add first record and enter data in kind, name, and ct*pair fields;
        otherwise will call add*to*sr*list to check existing list}
   var
        sr*temp : subrout*ptr;
begin
        case sr*kind of
          proc:
            begin
              if design*data*.proc*link = nil then
                begin
                      new (sr*data);
                      design*data*.proc*link := sr*data;
                      with sr*data* do
                        begin
                              kind := sr*kind;
                              name := sr*name;
                              ct*pair := nil;
                              stmts*link := nil;
```

```
                                next := nil;
                            end;  {with}
                    end
                  else
                    begin
                        sr^temp := design^data^.proc^link;
                        add^to^sr^list (sr^temp,name^used,sr^data,sr^kind,sr^name);
                    end;

              end;
            func:
              begin
                if design^data^.func^link = nil then
                  begin
                        new (sr^data);
                        design^data^.func^link := sr^data;
                        with sr^data^ do
                          begin
                                kind := sr^kind;
                                name := sr^name;
                                ct^pair := nil;
                                stmts^link := nil;
                                next := nil;
                          end;  {with}
                  end
                else
                  begin
                        sr^temp := design^data^.func^link;
                        add^to^sr^list (sr^temp,name^used,sr^data,sr^kind,sr^name);
                  end;
              end;
          end;  {case}
end;  {add^sr^record}


PROCEDURE enter^subroutine^name (var sr^data : subrout^ptr;
        var name^used : boolean; sr^kind : subrout^kind;
        design^data : design^ptr);
        {will read name entered, check to see if it has been previously used,
        and, if not, will add a new record to the appropriate subroutine list
        and enter the name, the kind, and set the c/t link to nil}
   var
        sr^name : name^size;

begin
        enter^item (sr^name);
        name^used := false;
        check^other^sr^list (name^used,sr^kind,design^data,sr^name);
        if not name^used then
          add^sr^record (name^used,sr^data,sr^kind,design^data,sr^name);
end;  {enter^subroutine^name}


        {STATEMENT ROUTINES}


PROCEDURE enter^variable^name (var vble^name : name^size);
        {will display menu and call enter^item to get
        variable name}
```

```
begin
        statements^menu^variable^name;
        draw^marker (1);
        enter^item (vbl^name);
end;  {enter^variable^name}



PROCEDURE link^variable (sr^data : subrout^ptr; vbl^rec : expr^item^ptr;
        var last^vbl : expr^item^ptr;
        var expr^set^link : expr^item^ptr);
        {will link latest terminal record in an expression to the
        list of variables in a subroutine, and then update the forward and
        backward pointers in that list}
begin
        if sr^data^.first^vbl = nil then
          begin
                sr^data^.first^vbl := vbl^rec;
                vbl^rec^.prev^vbl := nil;
          end
        else
          begin
                last^vbl^.next^vbl := vbl^rec;
                vbl^rec^.prev^vbl := last^vbl;
          end;
        vbl^rec^.expr^set := expr^set^link;
        last^vbl := vbl^rec;
end;  {link^variable}



PROCEDURE link^statement (sr^data : subrout^ptr; statement^data : stmt^ptr;
        var statement^ptr : stmt^ptr);
        {will link new statement to linked list of statements in a subroutine}
begin
        if sr^data^.stmts^link = nil then
           sr^data^.stmts^link := statement^data
        else
           statement^ptr^.next^stmt := statement^data;
        statement^ptr := statement^data;
end;  {link^statement}



PROCEDURE statement^io^mechanics (sr^data : subrout^ptr; kind : operators;
        var statement^ptr : stmt^ptr; var last^vbl : expr^item^ptr);
        {will obtain new record, enter type, obtain non-terminal record
        and enter kind of i/o statement, and then obtain a terminal
        record, enter the name, and call link^variable and link^statement
        to link up the statement and variable to previous ones in this subroutine}
   var
        stmt^data : stmt^ptr;
        io^rec^one, io^rec^two : expr^item^ptr;
        name : name^size;
begin
        new (stmt^data);
        stmt^data^.stmt^type := io;
        new (io^rec^one);
        stmt^data^.direction := io^rec^one;
        io^rec^one^.item := nonterm;
```

```
        io^.rec^.one^.operator := kind;
        new (io^.rec^.two);
        io^.rec^.one^.right^.ptr := io^.rec^.two;
        io^.rec^.one^.left^.ptr := nil;
        enter^variable^name (name);
        io^.rec^.two^.name := name;
        link^variable (sr^.data,io^.rec^.two,last^.vbl,io^.rec^.one);
        link^statement (sr^.data, stmt^.data, statement^.ptr);
end;  {statement^.io^.mechanics}


PROCEDURE enter^.io^.stmt (sr^.data : subrout^.ptr; var statement^.ptr : stmt^.ptr;
        var last^.vbl : expr^.item^.ptr);
        {will show input/output statement menu and call statement^.i/o^.mechanics with kind either
        data^.in or data^.out.  User also has the option of selecting
        'no op' and exiting without writing a new statement}
   var
        choice : char;
        done^.io : boolean;
begin
        done^.io := false;
        statements^.menu^.io;
        repeat  {until done^.io}
           clear^.input;
           readln (choice);
           clear^.msg;
           case choice of
             '1':       {input}
                begin
                  statement^.io^.mechanics (sr^.data,data^.in,statement^.ptr,last^.vbl);
                  done^.io := true;
                end;
             '2':       {output}
                begin
                  statement^.io^.mechanics (sr^.data,data^.out,statement^.ptr,last^.vbl);
                  done^.io := true;
                end;
             '3':
                begin
                  done^.io := true;
                  if sr^.data^.stmts^.link = nil then
                        statement^.ptr := nil;
                  if sr^.data^.first^.vbl = nil then
                        last^.vbl := nil;
                end;
             otherwise
                message (1);
           end;  {case}
        until done^.io;
end;  {enter^.io^.stmt}


PROCEDURE enter^.statements (sr^.data : subrout^.ptr);
        {will continue in a loop of displaying the types of statements
        available and calling the appropriate subroutines to enter
        them as the user enters menu choices until the user decides to exit}
   var
        done^.stmts : boolean;
```

```
        choice : char;
        statement_ptr : stmt_ptr;
        last_vbl : expr_item_ptr;
        line : integer;
begin
        clear_display;
        line := 1;
        display_subroutine_name (line, sr_data);
        display_begin (line);
        done_stmts := false;
        repeat  {until done_stmts}
           statements_menu_type;
           clear_input;
           readln (choice);
           clear_msg;
           case choice of
             '1':       {assignment stmt}
                message (3);
             '2':       {conditional stmt}
                message (3);
             '3':
                begin
                  enter_io_stmt (sr_data,statement_ptr,last_vbl);
                  display_io_stmt (line, statement_ptr);
                end;
             '4':       {exit}
                begin
                  statement_ptr.next_stmt := nil;
                  last_vblt.next_vbl := nil;
                  done_stmts := true;
                  display_end (line);
                end;
           end;  {case}
        until done_stmts;
end;  {enter_statements}



PROCEDURE enter_subroutine (design_data : design_ptr;
                sr_kind : subrout_kind);
        {will write menu for subroutine name and comments and call enter_subroutine_name
        to check whether the name entered has been used previously.  if not,
        will then call enter_comments to obtain comments}
    var
        sr_data : subrout_ptr;
        name_used : boolean;

begin
        subroutine_menu;
        draw_marker (1);
        enter_subroutine_name (sr_data,name_used,sr_kind,design_data);

        if not name_used then
          begin
                clear_input;
                clear_marker (1);
                draw_marker (2);
                enter_comments (sr_data.f,cont_link);
```

```
                enterstatements (srdata);
                entervariables (designdata, srdata);
                displaysubroutine (srdata);
                delay;
            end;
    end;  {enteresubroutine}
```

```
{file convert.pas - routines to convert run time data structures
to primitive list format}

{ALGORITHM check+exit (design+data : design+ptr; var done : boolean);
        {will check to see that all necessary data has been entered
        before data is converted to primitive list format}
begin
        If design +data+.id+link = nil then
          begin
                message 'data needed'
                enter+id (design+data+.id+link)
          end

        if design+data+.crit +link = nil then
          begin
                message 'data needed'
                enter+criteria (design+data+.crit+link)
          end

        if design+data+.ct+type+.run+list = nil
          begin
                message 'data needed'
                enter+ct+pairs (design+data+.ct+type+.run+list);
          end;
        {may want to repeat above for init+list also}

        if design+data+.func+link = nil then
          begin
                message 'data needed'
                enter+subroutine (design+data+.func+link)
          end
        {repeat above for procedure link}

        check := design+data+.func+link
        repeat {until check = nil}
                if check+.stmt+link = nil then
                  begin
                        message 'statements needed'
                        enter+subroutine (check)
                  end
                if check+.ct+pair = nil then
                  begin
                        messsage 'c/t data needed'
                        enter+ct+pair
                  end
                check := check+.next
        until check = nil

        {repeat above for procedures}

        done := true
end ;check+exit}

{end of convert.pas}
```

```
{file files.pas - contains pascal routines to write data
        entered to a file in CSD/Ada format}


{BASIC ROUTINES}

PROCEDURE open+file;
        {will open a file (with variable name data and prepare
        it for rewriting with new data}

begin
        open (data, 'data.csd');
        rewrite (data);
end; {open+file}



PROCEDURE close+file;
        {will close file named data and write it to memory}

begin
        close (data, disposition := save);
end; {close+file}



PROCEDURE file+comments (c : comt+ptr);
        {will traverse a linked list of lines of
        comments, writing the comments to the data file}

begin
        while c <> nil do
          begin
            writeln (data,'-- ', cf.line);
            c := cf.next+comt;
          end; {while}
end;  {file+comments}


{IDENTIFICATION DATA}

PROCEDURE file+id (id+data : id+ptr);
        {will write contents of the fields of the id
        record to the data file, and then will traverse
        the list of comment records and write the comments
        to the data file}

        var
                c : comt+ptr;


begin
        writeln (data, '--IDENTIFICATION DATA');
```

```
        writeln(data);
        if id^.data = nil then
                writeln (data,'-- no data entered')
        else
          with id^.data^ do
              begin
                 writeln (data,'-- Name:      ',designer);
                 writeln (data,'-- Project:   ',project);
                 writeln (data,'-- Date:      ',date);
                 writeln (data);
                 file^.comments (com^.link);

              end;  {with}
        writeln (data); writeln (data);

end; {file^.id}


{CRITERIA DATA}


PROCEDURE file^.criteria (crit^.data : criteria^.ptr);
        {will write heading and data contained in criteria, monitors,
        and volumes records}
begin
        writeln (data); writeln (data);
        writeln (data,'-- CRITERIA DATA');
        writeln (data);

        if crit^.data = nil then
          writeln (data,'-- no data entered')
        else
          begin
            with crit^.data^ do
              begin
                writeln (data,'-- Criteria:  ',crit);
                file^.comments (com^.link);
                with monitors^.link^ do
                  writeln (data,'-- monitors: ',m1:4,m2:4,m3:4,m4:4,m5:4);
                with volumes^.link^ do
                 begin
                  write (data,'-- volumes:  ',v1:4,v2:4,v3:4,v4:4,v5:4);
                  writeln (data, v6:4,v7:4,v8:4,v9:4,v10:4);
                 end;
              end;
          end;
        writeln (data); writeln (data);
end;  {file^.criteria}



{CONTINGENCY/TASK DATA}

PROCEDURE ct^.heading;

        {will write heading for table to display c/t data}
begin
```

```
        write (data,'-- CONTINGENCY      TASK        DEF     RHO   UNITS   BETA1  BETA2  GLOBAL  LOCAL  GAMMA1  GAMMA2   BGD    COND
        writeln (data);
end;  {ct+heading}



PROCEDURE file+ct+record (ct+ptr : ct+pair+ptr);
        {will write out contents of one contingency/pair record}

begin
        writeln (data);
           with ct+ptr^ do
             begin
                write (data,'-- ',funct.name,'    ');
                write (data, proct.name,'  ');
                write (data, cont+def,'  ');
                write (data, rho:6,'  ');
                write (data, time+units,'  ');
                write (data, beta1:6,'  ');
                write (data, beta2:6,'  ');
                write (data, glob+ord:6,'  ');
                write (data, loc+pri:6,'  ');
                write (data, gamma1:6,'  ');
                write (data, gamma2:6,'  ');
                write (data, background,'  ');
                write (data, ct+cond,'  ');
                writeln (data);
                file+comments (comt+link);
             end;
end;  {file+ct+record}



PROCEDURE file+ct+pairs (design+data : design+ptr);

        {will write title and then call file+ct+record repeatedly to write
        out contingency/task data}
   var
        temp : ct+pair+ptr;

begin
   writeln (data); writeln (data);
   writeln (data, '-- CONTINGENCY/TASK PAIR DATA');
   writeln (data);
   if ((design+data^.ct+link^.init+list = nil) and
       (design+data^.ct+link^.run+list = nil)) then
     writeln (data,'-- no data entered')
   else
     begin
        ct+heading; writeln (data);
        writeln (data,'-- Initialization');
        writeln (data);


        temp := design+data^.ct+link^.init+list;
        if temp = nil then
                writeln (data,'-- no data entered')
        else
```

```
          repeat  {until temp = nil}
                file^.ct^.record (temp);
                temp := temp^.next^.pair;
          until temp = nil;

        writeln (data); writeln (data);
        writeln (data,'-- Run time');
        writeln (data);

        temp := design^.data^.ct^.link^.run^.list;
        if temp = nil then
                writeln (data, '-- no data entered')
        else
          repeat  {until temp = nil}
                file^.ct^.record (temp);
                temp := temp^.next^.pair;
          until temp = nil;
        writeln (data);  writeln (data);
      end;  {else}
      writeln (data);  writeln (data);

end;  {file^.ct^.pairs}



        {VARIABLE DATA}


PROCEDURE file^.ext^.float^.vbl (var^.data : vbl^.ptr);
        {will write data entered for a variable of type external
        floating point to a file}
begin

   writeln (data,'--        value     addr      prec       min       max      mantis    expon      port');
   with var^.data^ do
    begin
     write (data,'-- ',value,mem^.addr,precision^.efl,min^.val^.efl,max^.val^.efl);
     writeln (data,mantis^.efl,expon^.efl,port^.num^.efl);
    end;

end;  {file^.ext^.float^.vbl}



PROCEDURE file^.variables (var^.data : vbl^.ptr);
        {will traverse a list of variables, writing their name and
        type to a file, and then calling the subroutine appropriate to
        their type to write the rest of the information}
  var
        var^.temp : vbl^.ptr;
begin
        writeln (data); writeln (data);
        writeln (data,'-- VARIABLES');
        writeln (data);
        if var^.data = nil then
          writeln (data,'-- no variables entered')
        else
          begin
            var^.temp := var^.data;
```

```
            while var*temp <> nil do
              begin
                writeln (data, var*temp*.name,' : ',var*temp*.kind1,';');
                case var*temp*.kind1 of
                   (analog:
                        ();
                   booln:
                        ();
                   ext*fixed:
                        ();)
                   ext*float:
                        file*ext*float*vbl (var*temp);
                   (int*fixed:
                        ();
                   int*float:
                        ();)
                end;  (case)
                writeln (data);
                var*temp := var*temp*.next*vbl;
              end;  (while)
          end;  (else)
        writeln (data); writeln (data);
end;  (file*variables)
```



```
(FUNCTION AND PROCEDURE DATA)


PROCEDURE file*io*stmt (statement*ptr : stmt*ptr);
        (will write record data to file in io statement format)
    var
        kind : (sense, issue);
begin
        if statement*ptr <> nil then
          begin
            case statement*ptr*.direction*.operator of
                data*in:
                        kind := sense;
                data*out:       (output)
                        kind := issue;
            end;  (case)

            writeln (data,kind,'(',statement*ptr*.direction*.right*ptr*.name,');');
          end;
end;  (file*io*stmt)
```

```
PROCEDURE file^statements (statemts^ptr : stmt^ptr);
        {will traverse linked list of statements and write them
        to a file until it reaches the end of the list}
   var
        statement^ptr : stmt^ptr;
begin
        statement^ptr := statemts^ptr;
        if statement^ptr = nil then
          writeln (data,'-- no statements entered')
        else
         begin
          writeln (data,'BEGIN');
          repeat  {until statement^ptr = nil}
            case statement^ptr^.stmt^type of
                io:
                   file^io^stmt (statement^ptr);
                {add more later}
            end;  {case}
            statement^ptr := statement^ptr^.next^stmt;
          until statement^ptr = nil;
          writeln (data,'END;');
         end;
end;  {file^statements}
```



```
PROCEDURE file^subroutine (s^ptr : subrout^ptr);

        {will write subroutine data to file, calling file^comments and file^statements}

begin
        file^comments (s^ptr^.comt^link);
        file^variables (s^ptr^.loc^vbl^link);
        file^statements (s^ptr^.stmts^link);
        writeln (data); writeln (data);
end;


PROCEDURE file^functions (func^ptr : subrout^ptr);
        {will write heading and then traverse the function
        list, writing function names and calling
        file^subroutine to write remainder of data
        to file data}

        var
                f^temp : subrout^ptr;

begin
        f^temp := func^ptr;
        writeln (data); writeln (data);
        writeln (data,'-- FUNCTIONS');
        writeln (data);

        if f^temp = nil then
                writeln (data,'no functions entered')
        else
           repeat  {until f^temp = nil}
```

```
                writeln (data,'function ',f^temp^.name,';');
                file^subroutine (f^temp);
                f^temp := f^temp^.next;
            until f^temp = nil;
        writeln (data); writeln (data);

end;  {file^functions}

PROCEDURE file^procedures (proc^ptr : subrout^ptr);
        {will write heading and then traverse the procedure
        list, writing procedure names and calling
        file^subroutine to write remainder of data
        to file data}

        var
                p^temp : subrout^ptr;

begin
        p^temp := proc^ptr;
        writeln (data); writeln (data);
        writeln (data,'-- PROCEDURES');
        writeln (data);

        if p^temp = nil then
                writeln (data,'no procedures entered')
        else
           repeat  {until p^temp = nil}
                writeln (data,'procedure ',p^temp^.name,';');
                file^subroutine (p^temp);
                p^temp := p^temp^.next;
           until p^temp = nil;
        writeln (data); writeln (data);

end;  {file^procedures}


{MAIN ROUTINE}


PROCEDURE file^main (design^data : design^ptr);
        {will open file, call subroutines to write sections of data to
        the file, close the file, and write a message to the screen
        that the data has been written to a file}

begin
        open^file;
        file^id (design^data^.id^link);
        file^criteria (design^data^.crit^link);
        file^ct^pairs (design^data);
        file^variables (design^data^.env^link);
        file^functions (design^data^.func^link);
        file^procedures (design^data^.proc^link);
        close^file;
        message^posit;
        writeln ('!t(s1,w(v,i(c)))''data written to file');
end; {file^main}


{end files.pas}
```

```
{file test.pas - will be used to call routines being tested}



procedure test;
        var
                design_data : design_ptr;
                item : name_size;
                line : integer;
                variable_rec : vbl_ptr;
                variable : expr_item_ptr;
                vbl_1,vbl_2 : expr_item_ptr;
        last_vbl, prev_vbl : expr_item_ptr;
                sr_rec : subrout_ptr;
                statement_ptr : stmt_ptr;


begin
        main;


end;  {test}
```

```
{file change.pas - routines for making changes to data
already entered}

PROCEDURE change_digits (var value : integer);
        {will read and check length of input.  If length > 0, will call
        check_digits to check entry for other criteria.  If not
        errors are found, will substitute new value for old.
        Is called with pointer to an already filled record field as
        its parameter}

   var
        temp : digits;
        done_digits : boolean;
        con_val : integer;
begin
        done_digits := false;
        repeat  {until done_digits}
           readln (temp);
           if length (temp) > 0 then
             begin
                check_digits (con_val, temp, done_digits);
                if done_digits then
                   value := con_val;
             end
           else
             done_digits := true;
        until done_digits;
end;  {change_digits}



PROCEDURE change_ct_values(ct_data : ct_pair_ptr);
        {will show menu for changing values and call change_digits to
        check entries for each value in the list}

begin
        ct_change_menu_values;

        draw_marker (1);
        change_digits (ct_data^.rho);
        clear_marker (1);

        draw_marker (2);
        change_digits (ct_data^.beta1);
        clear_marker (2);

        draw_marker (3);
        change_digits (ct_data^.beta2);
        clear_marker (3);

        draw_marker (4);
        change_digits (ct_data^.glob_ord);
        clear_marker (4);

        draw_marker (5);
        change_digits (ct_data^.loc_pri);
        clear_marker (5);
```

```
        draw_marker (6);
        change_digits (ct_dataf.gamma1);
        clear_marker (6);

        draw_marker (7);
        change_digits (ct_dataf.gamma2);
        clear_marker (7);

end;  {change_ct_values}



PROCEDURE change_ct_data (ct_data : ct_pair_ptr);
        {will display menu of areas for change of ct data and continue in
        loop of calling specific change routines until user desides
        to exit}
   var
        choice : char;
        done_change : boolean;
begin
        done_change := false;

        repeat  {until done_change}
           clear_input;
           ct_change_menu_data;
           readln (choice);
           clear_msg;
           case choice of
             '1':       {timing units}
                message (3);
             '2':       {contingency data}
                message (3);
             '3':       {data values}
                change_ct_values(ct_data);
             '4':       {background}
                message (3);
             '5':       {condition}
                message (3);
             '6':       {comments}
                message (3);
             '7':       {exit}
                done_change := true;
             otherwise
                message (1);
           end;  {case}
           display_ct_pair_data (ct_data);
        until done_change;
end;  {change_ct_data}



PROCEDURE change_ct_pairs (ct_data : ct_pair_ptr);
        {will display menu and determine whether user desires to make
        changes in linkage of c/t pairs (procedure names or type of
        pair, initialization or run time), or in data assigned to various
        fields of the c/t pair record.  Will continue in loop until user
        desires to exit}
   var
        choice : char;
```

```
        done_change : boolean;
begin
        done_change := false;
        display_ct_pairs_data (ct_data);
        repeat  {until done_change}
          clear_input;
          ct_change_menu_areas;
          readln (choice);
          clear_msg;
          case choice of
            '1':        {linkage}
                message (3);
            '2':        {data}
                change_ct_data (ct_data);
            '3':        {exit}
                done_change := true;
            otherwise
                message (1);
          end;  {case}
        until done_change;
end;  {change_ct_pairs}



PROCEDURE change_main (design_data : design_ptr);
        {main routine for making changes to entire set of already entered
        data.  Will display menu of work areas and call change subroutines
        based on choice entered by user.  Will continue in loop
        until user decided to exit.}
    var
        done_change : boolean;
        choice : char;
        temp : ct_pair_ptr;
begin
        done_change := false;
        repeat  {until done_change}
          clear_input;
          main_change_menu;
          readln (choice);
          clear_msg;
          case choice of
            '1':        {id}
                message (3);
            '2':        {criteria}
                message (3);
            '3':        {c/t pairs}
                begin
                  temp := design_data^.ct_link^.init_list;
                  while temp <> nil do
                    begin
                        change_ct_pairs (temp);
                        clear_display;
                        temp := temp^.next_pair;
                    end;
                  temp := design_data^.ct_link^.run_list;
                  while temp <> nil do
                    begin
                        change_ct_pairs (temp);
                        clear_display;
                        temp := temp^.next_pair;
```

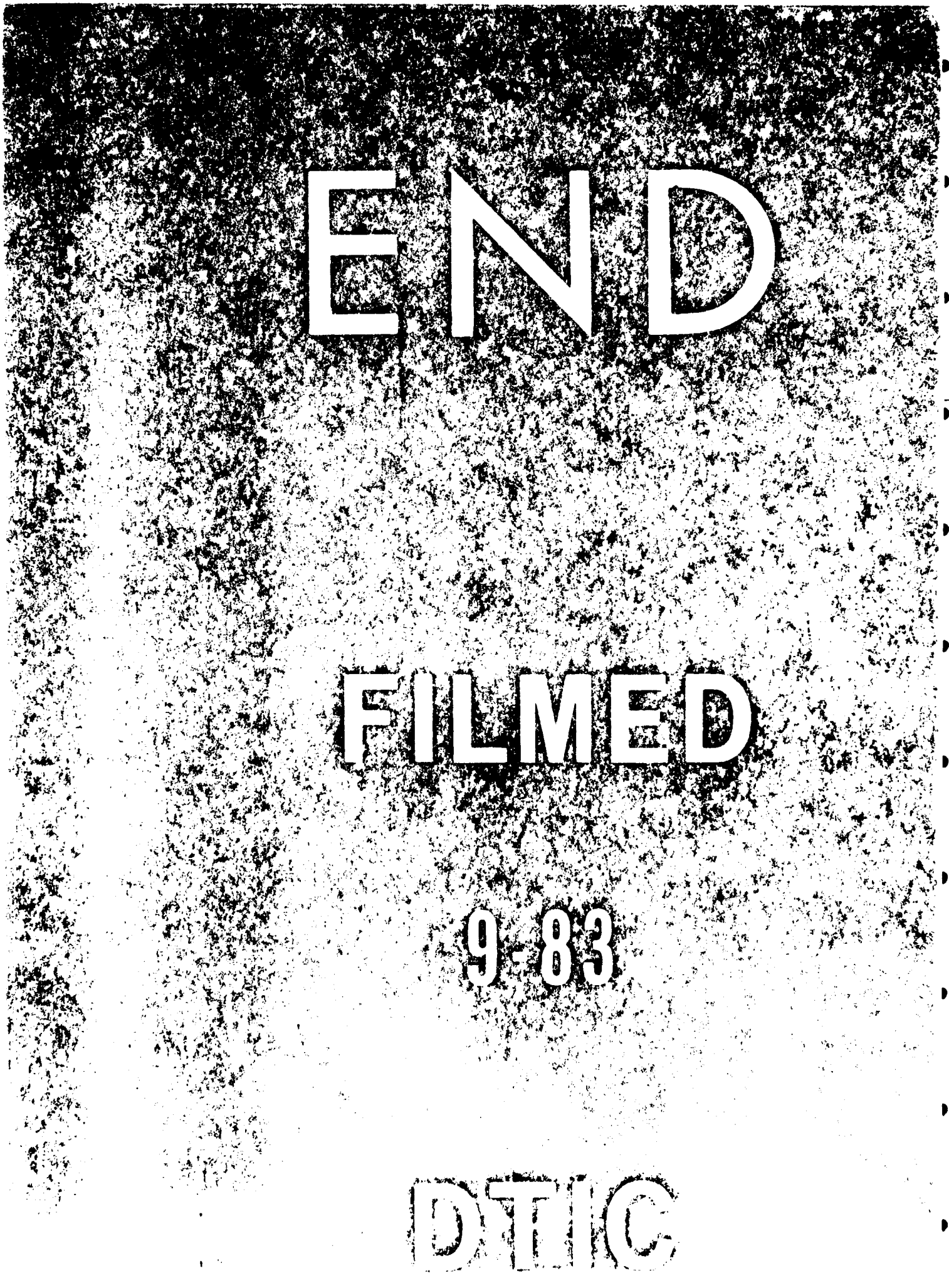
```
                    end;

                end;
            '4':        {procedures}
                message (3);
            '5':        {functions}
                message (3);
            '6':        {exit}
                done_change := true;
            otherwise
                message (1);
        end;  {case}
    until done_change;
end;  {change_main}


{end file change.pas}
```

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0412 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Curricular Officer Computer Technology, Code 37 Naval Postgraduate School Monterey, California 93940	1
5.	LTCOL Alan A. Ross, USAF Department of Computer Science, Code 52Rs Naval Postgraduate School Monterey, California 93940	2
6.	Professor George A. Rahe Department of Computer Science, Code 52Ra Naval Postgraduate School Monterey, California 93940	1
7.	Professor Herschel H. Loomis Department of Electrical Engineering, Code 62Lm Naval Postgraduate School Monterey, California 93940	1
8.	Ms. Jeanne L. Bowers Department of Computer Science, Code 52 Naval Postgraduate School Monterey, California 93940	1
9.	Mr. Joel Trimble Code 240 Office of Naval Research 800 North Quincy Street Arlington, VA 22217	1
10.	Professor R. Steven Schiavo Psychology Department Wellesley College Wellesley, MA 02181	1

11. Library 1
Wellesley College
Wellesley, MA 02181

12. LCDR Barbara J. Sherlock, USN 2
EME 120 - 3
Naval Electronic Systems Command Headquarters
Washington, D. C. 20360


END
FILMED
9-83
DTIC